Catalog Number 26-2112 $5.95

TRS-80™ Model III

Operation and BASIC Language Reference Manual

```
870 '          ROUTINE #7
880 FOR I = 1 TO N1
890 FOR J = 1 TO N2
900 B(J,I) = A(I,J)
910 NEXT J
920 NEXT I
930 END

730   NEXT I
740 PRINT: NEXT
750 END

770 '        ROU
780 FOR K = 1
790 FOR J = 1 TO N2
800 FOR I = 1
810 B(I,J,K)
840 NEXT K
850 END

960 FOR K = 1 TO N3
970 FOR J = 1 TO N2
980 FOR I = 1 TO N1
990 C(I,J,K) = A(I,J,K) + B(I, J, K)
1000 NEXT I
1010 NEXT J
1020 NEXT K
1030 END
```

CUSTOM MANUFACTURED IN THE USA BY RADIO SHACK A DIVISION OF TANDY CORPORATION

TRS-80™ Model III

Operation and BASIC Language Reference Manual

Radio Shack®

A DIVISION OF TANDY CORPORATION
FORT WORTH, TEXAS 76102

TRS-80 Model III Operation and BASIC Language Reference Manual: ©1980 Tandy Corporation, Fort Worth, Texas 76102 U.S.A. All Rights Reserved.

Reproduction or use, without express written permission from Tandy Corporation, of any portion of this manual is prohibited. While reasonable efforts have been taken in the preparation of this manual to assure its accuracy, Tandy Corporation assumes no liability resulting from any errors or omissions in this manual, or from the use of the information obtained herein.

Model III System Software: ©1980 Tandy Corporation and Microsoft. All Rights Reserved.

The system software in the Model III microcomputer is retained in a read-only memory (ROM) format. All portions of this system software, whether in the ROM format or other source code form format, and the ROM circuitry, are copyrighted and are the proprietary and trade secret information of Tandy Corporation and Microsoft. Use, reproduction or publication of any portion of this material without the prior written authorization by Tandy Corporation is strictly prohibited.

10 9 8 7 6 5 4 3 2

To Our Customers. . .

The TRS-80™ Model III Computer is a very powerful tool for business, home and recreation. Twenty years ago, this capability would have cost hundreds of times as much as your Model III cost, and would have taken up an entire room.

In spite of its power and internal complexity, the Model III can be quite simple to operate. In fact, **you** can determine just how "technical" a machine you want it to be.

At the simplest level of operation, you can use Radio Shack prepared cassette programs. All you will need to know is how to load and run a cassette program, and how to operate the cassette recorder. If this is where you want to start, read Chapters 1 through 6 of the Operation Section. You may also want to read about CLOAD and SYSTEM in Chapter 2 of the Language Section.

If you want to write your own programs and you are a beginner, read Chapters 1 through 6 of the Operation Section, then start reading the book, *Getting Started with TRS-80 BASIC*. That, plus several other Radio Shack books, can guide you to becoming a programmer in BASIC and Z-80 language ("machine code").

If you already know BASIC, and especially if you have experience on a TRS-80 Model I, read the entire Operation Section of this manual, as well as the Appendix which compares the Model I and Model III. The Model III has many unique features and some very important differences. A few minutes spent before you press (ENTER) could save you hours later.

About This Manual

This manual contains operating instructions (Section 1) and a description of Model III BASIC (Section 2 and Appendix). It is arranged for easy reference, whether you are seeking simple or technical information. Page numbering starts over at the beginning of each chapter, and chapter numbering starts over at the beginning of each section. There is a comprehensive Index at the end of this book.

If you are a beginner, don't worry about the technical parts in the Operation Section. The beginning of each chapter is for you. (When you get to the POKE statements, you can skip ahead to the next chapter...) You don't need to read past Chapter 6. Then, when you learn simple BASIC programming, you can return and try out all the "goodies" packed into your Model III.

Very Important Note

Before you even plug in your Model III, read Chapters 2 and 3 in the Operation Section—no matter how much you think you know. This applies whether you have a cassette- or disk-based system.

Remember, when all else fails, read the instructions!

Contents

Operation Section

A Brief Description of the Computer **1**/1-3

Installation **2**/1-3

Operation **3**/1-9
Power-On ☐ RESET Switch ☐ Power-Off ☐ Start-Up Dialog ☐ Modes of Operation ☐ Sample Session

Using the Keyboard **4**/1-3
Capitals and Lowercase ☐ Special Keys ☐ Control Codes

Using the Video Display **5**/1-5
Character Size ☐ Cursor ☐ Scroll Protection ☐ Text ☐ Graphics ☐ Space Compression ☐ Special Characters

Using the Cassette Interface **6**/1-6
Cassette Transfer Speed ☐ Loading Errors ☐ Saving a BASIC Program on Tape ☐ Loading a BASIC Program from Tape ☐ How to Search for a Program ☐ Loading a SYSTEM Tape ☐ Searching for a Program

Using a Line Printer **7**/1-6
Line Printer *vs* Video Display Output ☐ Printer Control Features ☐ Print Screen Function

Using the RS-232-C Interface **8**/1-8
What is an Interface? ☐ Using the Model III as a Terminal ☐ Programming the RS-232-C

Routing Input/Output **9**/1-3
To Route from One Device to Another ☐ Routing Multiple Devices

Real-Time Clock **10**/1-3
To Set the Clock ☐ To Read the Clock ☐ To Display the Clock

Input/Output Initialization **11**/1-1

Technical Information **12**/1-26
To Protect High RAM ☐ ROM Subroutines ☐ Memory Map ☐ Summary of Important ROM Addresses ☐ Summary of Important RAM Addresses

Troubleshooting and Maintenance **13**/1-3
Symptom/Cure Table ☐ AC Power Sources ☐ Maintenance

Specifications **14**/1-3
Power Supply ☐ Microprocessor ☐ RS-232-C Interface ☐ Parallel (Printer) Interface ☐ Cassette Interface

BASIC Language Section

BASIC Concepts **1**/1-30

Commands **2**/1-7

Input-Output Statements **3**/1-13

Program Statements **4**/1-15

Strings **5**/1-9

Arrays **6**/1-6

Arithmetic Functions **7**/1-5

Special Features **8**/1-10

Editing **9**/1-7

Appendices

Model III Summary **A**/1-18

Special Characters and Abbreviations ☐ Commands ☐ Statements ☐ Functions ☐ Reserved Words ☐ Program Limits ☐ Memory Use ☐ Accuracy ☐

Error Codes **B**/1-3

TRS-80 Model III Character Codes **C**/1-9

Keyboard/Display Characters ☐ Graphics ☐ Special Characters ☐ Video Display Worksheet ☐

Internal Codes for BASIC Keywords **D**/1-2

Derived Functions **E**/1-2

Base Conversions **F**/1-4

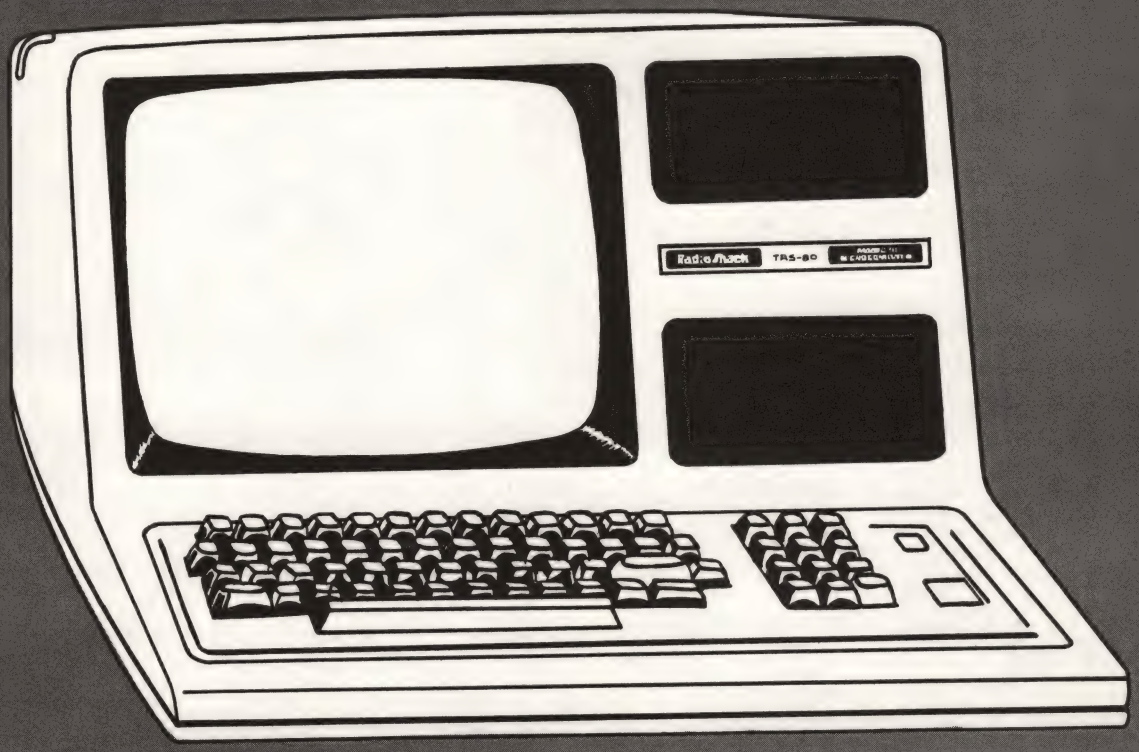
Model I to Model III Program Conversion Hints **G**/1-2

Glossary **H**/1-3

RS-232-C Technical Information **I**/1-4

Index

For Warranty and Customer Information, see the back cover and inside back cover.



Section 1: Operation

1 / A Brief Description

The Radio Shack TRS-80™ Model III is a ROM-based computer system consisting of:

- A 12-inch screen to display results and other information
- A 65-key console keyboard for inputting programs and data to the Computer
- A Z-80 Microprocessor, the "brains" of the system
- A Real-Time Clock
- Read Only Memory (ROM) containing the Model III BASIC Language (fully compatible with most Model I BASIC programs)
- Random Access Memory (RAM) for storage of programs and data while the Computer is on (amount is expandable from "16K" to "48K", optional extra)
- A Cassette Interface for long-term storage of programs and data (requires a separate cassette recorder, optional/extra)
- A Printer Interface for hard-copy output of programs and data (requires a separate line printer, optional/extra)
- Expansion area for upgrading to a disk-based system (optional/extra)
- Expansion area for an RS-232-C serial communications interface (optional/extra)

All these components are contained in a single molded case, and all are powered via one power cord.

Video Display Screen

Displayable characters include the standard 96 text-characters with the upper and lowercase alphabet; 64 graphics characters; and 160 special TRS-80 characters. In addition, there are numerous control and space-compression characters. Some of the character sets can be switched in and out by BASIC and other programs.

Keyboard

The keyboard allows entry of all the standard text and control characters. It also includes a 12-key section for convenient numeric entry. From the keyboard, you can select either all-capitals or upper and lowercase entry. The (BREAK) key is designed to return control to you during any operation, including cassette input/output or line printer output. Every key has an auto-repeat feature.

Z-80 Microprocessor

This is the central processing unit—where all the ''thinking'' is done. In the Model III, the microprocessor operates at a speed of over two million cycles per second.

Read Only Memory (ROM)

This is where the Computer's built-in programs are stored, including the TRS-80 BASIC language. TRS-80 BASIC is fully compatible with the Level II language used in Model I TRS-80's. Each time you power-on the Computer, this ROM program takes charge of the microprocessor, enabling you to type in simple BASIC-language instructions.

The Model III contains a ''14K'' ROM, meaning it contains 14 * 1024 = 14336 characters (''bytes'') of permanently programmed memory.

Random Access Memory (RAM)

This is where your programs and results are stored while the Computer is on. It is erased when you turn the Computer off.

The Model III can be equipped with 16K, 32K or 48K of RAM (1K = 1024 bytes).

Peripherals

These are devices you can add to your Computer to increase its usefulness in programming and data storage. The Model III contains the necessary ''interfaces'' to simplify the addition of many peripherals.

Cassette

For long-term storage of programs and data, simply connect a cassette recorder to the Computer, and save the information on tape.

For program storage, you may select either High or Low transfer rates (use Low for compatibility with Model I, High for faster saves and loads).

Printer

You may connect any Radio Shack ''parallel interface'' printer to the Model III; this will give you ''hard-copy'' capability for program listings, reports, mailing lists, invoices, etc.

Other Enhancements

The Model III contains space for a mini-disk controller and one or two mini-disk drive units. The Computer will accommodate one or two external drive units as well.

With a one-, two-, three- or four-drive system, you will be able to store and retrieve programs and data both quickly and reliably. Your Computer will then be under the control of TRSDOS™, the powerful Radio Shack Disk Operating System.

You can also add an internal RS-232-C serial interface. This will allow your computer to communicate with an RS-232-C equipped computer, serial line printer or other serial device.

2 / Installation

Carefully unpack the Computer. Remove all packing material and save it in case you ever need to transport the Computer. Be sure to locate all cables, papers, etc., that may be included in the shipping carton.

Place the Computer on the surface where you'll be using it. An appropriate power source should be nearby, so that no extension cord will be required.

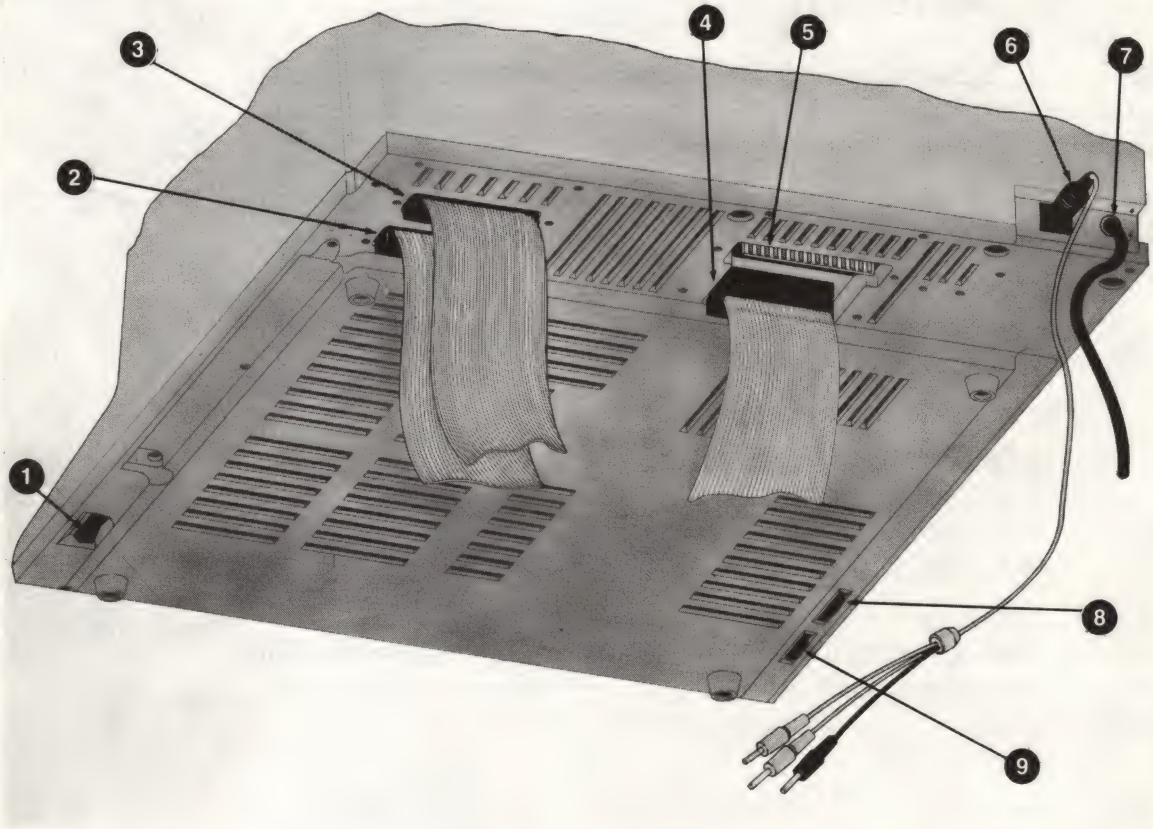
Do not connect the Computer to the AC power source yet.

Connection of Peripherals

Before connecting any peripherals (for example, line printer and cassette recorder), make sure the Computer and the peripheral devices are turned off.

Connect all peripherals to the appropriate jacks on the bottom and rear of the Computer. Refer to Figure 1 for location of connection points. For interconnections between cables and peripherals, refer to the Owner's Manual supplied with the peripheral device.

Note: All cables should exit to the rear of the unit so that no binding occurs.



1. **On/Off Switch**
2. **Disk Expansion Jack.** Shown with cable connected. Cable is supplied with external drives (optional/extra).
3. **Parallel Printer Jack.** Shown with cable connected. Cable and printer are optional/extra.
4. **RS-232-C Jack.** Shown with cable connected. Cable and RS-232-C Interface are optional/extra.
5. **I/O Bus Jack.** For future expansion.
6. **Cassette Jack.** Shown with cable connected. Cable and cassette recorder are optional/extra. Black mini-plug connects to recorder EAR; gray mini-plug to recorder AUX; gray submini-plug to MIC REMote control.
7. **AC Power Cord.**
8. **Video Contrast Adjustment.**
9. **Video Brightness Adjustment.**

Figure 1. Connection of peripherals and location of controls.

Connection of a Cassette Recorder

The following instructions use the CTR-80A recorder (Radio Shack Catalog Number 26-1206) as an example. If you use a different recorder, connection and operation may vary.

Note: You do not need to connect the Cassette Recorder unless you plan to record programs or to load taped programs into the TRS-80.

A TRS-80 to Cassette Recorder connection cable is included with the CTR-80A; we suggest that you use this specially designed cable.

1. Connect the short cable (DIN plug on one end and three plugs on the other) to the **TAPE** jack on the back of the Computer. **Be sure you get the plug to mate correctly.**
2. The three plugs on the other end of this cable are for connecting to the recorder.
3. A. Connect the **black plug** into the EAR jack on the side of the recorder. This connection provides the output signal from the recorder to TRS-80 (for loading Tape programs into TRS-80).

   B. Connect the larger **gray plug** into the AUX jack on the recorder. This connection provides the recording signal to record programs from the TRS-80 onto the tape.

   Leave the AUX plug in whether you are recording or playing back cassette data.

   C. Connect the smaller gray plug into the smaller MIC jack on the recorder. This allows the TRS-80 to automatically control the recorder motor (turn tape motion on and off for recording and playing tapes.)

Note: Do not plug a remote microphone or a dummy plug into the larger MIC jack.

Connection to an AC Power Source

Make sure the Computer and all peripherals are **off**.

The AC Power Cord exits from the rear of the Computer. Connect it and all peripherals to an appropriate power source. Power requirements for Radio Shack products are specified on the units and in the Owner's Manual Specifications.

For convenience, you may connect all components to a single "power strip" such as Radio Shack's 26-1451 Line Filter. This will allow you to turn on the entire system with a single switch. Take care not to exceed the current capacity of the power strip.

3 / Operation

Power-On

The following instructions explain how to start up and use the Model III **as a ROM-based system only.**

If you have a Disk System and are going to load TRSDOS, follow the power-up instructions given in the Model III Disk System Owner's Manual. If you have a Disk System but you are not going to load TRSDOS, read the instructions later in this chapter.

The Computer and all peripherals must be **off**.

First turn on all peripherals, then turn on the Computer. (If you have all the components connected to a power strip, just turn on the power strip.)

After a few seconds, the following message should appear on the Video Display:

```
Cass?
```

The meaning of this message will be explained later.

If the message does not appear:

A. The Video Display may need Brightness or Contrast adjustment. See Figure 1 for location of these controls.

B. If the message still doesn't appear, then turn off the entire system, recheck all connections, and try again. For further assistance, see "Troubleshooting and Maintenance."

Do not turn any peripherals off while the Computer is in use; to do so could cause abnormal operation (the Computer could restart or "hang up", requiring you to reset or turn the system off and on again).

RESET

RESET is the orange-colored button at the upper right corner of the keyboard. To "start over" at the power-on message, you do not have to turn the unit off and on again. Pressing the RESET button will have the same effect.

Note: Resetting the Computer does not erase the contents of RAM. However, the BASIC language interpreter will start over, thus "losing" any program or data you had in memory.

To interrupt a program or operation **without** losing your BASIC program and data, hold down the (BREAK) key.

Power-Off

First turn off the Computer, then all other peripherals.

If you turn the Computer off for any reason, leave it off for at least 15 seconds before turning it back on again. The Computer's power supply needs this time to discharge its stored energy before starting up again.

Whenever you turn off the Computer, all programs and data are erased. So be sure to save your information (e.g., on cassette) before turning off the Computer.

Start-Up Dialog

When you turn on or reset the Computer, it asks you two questions. First:

```
Cass?
```

This question lets you determine the rate at which programs and data will be transferred to and from cassette. You can select either Low (500 baud) or High (1500 baud). Type

```
L
```

for Low, or

```
H
```

for High.

If you press (ENTER) without typing anything, High will be used.

For further details, see ''Using the Cassette Interface.''

Next the Computer will ask:

```
Memory Size?
```

This question lets you set an upper limit to the RAM which will be used to store and execute your BASIC programs. Simply press (ENTER) in response to this question. This tells the Computer to make the full amount of RAM available for use by your BASIC program.

Advanced programmers may want to reserve some memory for a machine-language ("Z-80") program or subroutine. Instructions for doing this are included in the ''Technical Information'' chapter.

After you respond to the ''Memory Size'' question, BASIC will start with this message:

```
Radio Shack Model III Basic
(c) '80 Tandy
READY
>
```

The Computer is now ready for use.

Special Instructions for Disk System Owners Using Model III without TRSDOS

If you have a disk drive and disk controller installed, hold down the (BREAK) key **whenever you turn on or reset the Computer.** This tells the Computer that you are **not** going to use the disk capability.

Modes of Operation

BASIC has four modes of operation:

- Immediate mode—for typing in program lines and immediate lines
- Execute mode—for execution of programs and immediate lines
- Edit mode—for editing program and immediate lines
- System mode—for loading machine-language tapes and for transferring control to machine-language programs

Immediate Mode

Whenever you enter the immediate mode, BASIC displays a header and a special prompt:

READY	(header)
>■	(prompt followed by blinking block "cursor")

While you are in the immediate mode, BASIC will display the prompt at the beginning of the current logical line (the line you are typing in).

In the immediate mode, BASIC does not take your input until you complete the logical line by pressing (ENTER). This is called "line input", as opposed to "character input".

Interpretation of an Input Line

BASIC always ignores leading spaces in the line—it jumps ahead to the first non-space character. If this character **is not** a digit, BASIC treats the line as an immediate line. If it *is* a digit, BASIC treats the line as a program line.

For example:

```
PRINT "THE TIME IS"; TIME$ (ENTER)
```

BASIC takes this as an immediate line.

If you type:

```
10 PRINT "THE TIME IS"; TIME$ (ENTER)
```

BASIC takes this as a program line.

Immediate Line

An immediate line consists of one or more statements separated by colons. The line is executed as soon as you press (ENTER). For example:

```
CLS: PRINT "THE SQUARE ROOT OF 2 IS"; SQR(2)
```

is an immediate line. When you press (ENTER), BASIC executes it.

Program Line

A program line consists of a line number in the range [0,65529], followed by one or more statements separated by colons. When you press (ENTER), the line is stored in the program text area of memory, along with any other lines you have entered this way. The program is not executed until you type RUN or another execute command. For example:

```
100 CLS: PRINT "THE SQUARE ROOT OF 2 IS"; SQR(2)
```

is a program line. When you press (ENTER), BASIC stores it in the program text area. To execute it, type:

```
RUN (ENTER)
```

Special Keys in the Immediate Mode

(?) The question mark can stand for the commonly used keyword PRINT. For example, the immediate line:

```
? "HELLO."
```

is the same as the immediate line:

```
PRINT "HELLO."
```

Note: L? does *not* mean LPRINT.

This abbreviation can be used in a program, too.

(.) The period can stand for "current program line", i.e., the last program line entered or edited. The period can be used in most places where a line number would normally appear. For example, the immediate line:

```
LIST.
```

tells BASIC to list the current program line.

(') The single-quote tells BASIC to ignore the rest of the logical line. It is an abbreviation for the BASIC keyword REM. When used in a multi-statement line, it does not have to be preceded by a colon. For example, when you type in the line:

```
PRINT  1+1;  '2+2
```

BASIC will print the sum 1 + 1 but not 2 + 2.
This abbreviation can be used in a program, too.

(SHIFT)(↓)(*) Causes the Computer to print the Display contents to the line printer, if available. Press (BREAK) to interrupt this operation. This key sequence works in the other modes too.

Execute Mode

Whenever BASIC is executing statements (immediate lines or programs) it is in the execute mode. In this mode, the contents of the Video Display are under program control.

Special Keys in Execute Mode

(SHIFT) (@) Pauses execution. Press any key to continue.

(BREAK) Terminates execution and returns you to the command mode.

Edit Mode

BASIC includes a line editor for correcting program lines. To edit a program line, type in the command:

EDIT *line number*

where *line number* specifies the desired line.

When the editor is working on a program line, it displays the number of the line being edited.

In the edit mode, the Keyboard input is character-oriented, rather than line-oriented. That is, BASIC takes characters as soon as they are typed in—without waiting for you to press (ENTER).

See the chapter on editing (Section 2) for details.

System Mode

In this mode, you can load and execute machine-language programs. By "machine-language", we mean the set of machine instructions recognized by your Computer's Z-80 microprocessor. In this manual, we will usually call it "Z-80" programming, in contrast to BASIC programming.

You don't have to understand the Z-80 language to use some of the programs available. For example, several Radio Shack games are written in Z-80 code rather than in BASIC. To load such programs from tape, you use the System Mode.

Z-80 programming opens up whole new worlds of possibilities, but it is somewhat more demanding than BASIC programming.

The Technical Information chapter in this manual is written for those who are familiar with the Z-80 instruction set and other fundamental machine concepts. If you would like to explore these subjects, read:

TRS-80 Assembly Language Programming, by William Barden, Jr. Radio Shack Catalog Number 62-2006.

Although the book was originally written for the TRS-80 Model I, it applies almost exactly to the Model III as well.

For further details, see "Cassette Interface" in this Operation Section, and SYSTEM in the Language Section.

Sample Session

This section will give you a step-by-step example of what's needed to type in a program and run it. We will be showing you the Computer/operator dialog exactly as it appears on the Display. If you have never used a computer keyboard before, read **Using the Keyboard** before trying this sample session.

You don't need to know BASIC programming to go through this session—it is just an exerciser. If you are curious about the words used in this program, look them up on the Quick Reference Card supplied with your Computer, or in the Index of this manual.

Special Notation Used in this Dialog

BOLDFACE MATERIAL	Provided by the Computer—you don't type it in.
(ENTER)	Means "Press the (ENTER) key."
(SHIFT) (0)	This tells you to use the upper/lower case—caps only switch. You do this by pressing (SHIFT) and (0) together.
(→)	This means "press the (→) key" to skip over to the next eight-column boundary. We usually do this just for visual effect.

Answering the Start-Up Questions

Reset the Computer. Then follow this session.

Cass? (ENTER)

Memory Size? (ENTER)

Radio Shack Model III Basic

(c) '80 Tandy

READY

>■

The blinking block after ">" is the "cursor". It tells you where the next character you type will be displayed.

Now continue:

```
>NEW (ENTER)
READY
>AUTO (ENTER)
10 CLS (ENTER)
20 PRINT "HI—I'M YOUR TRS-80 MICROCOMPUTER!" (ENTER)
30 PRINT "(SHIFT) (0) What makes me so smart? "(SHIFT) (0) (ENTER)
40 PRINT "(SHIFT) (0) Millions of these: "(SHIFT) (0) (ENTER)
50 PRINT CHR$(21) (ENTER)
60 FOR I = 1 TO 256 (ENTER)
70 (→) PRINT CHR$(253); CHR$(254); (ENTER)
80 NEXT I (ENTER)
90 PRINT CHR$(21) (ENTER)
100 END (ENTER)
110 (BREAK)
READY
>■
```

Now the program is in memory. To look at it, type:

```
>LIST (ENTER)
```

It should look like this:

```
10 CLS
20 PRINT "HI! I'M YOUR TRS-80 MICROCOMPUTER!"
30 PRINT "What makes me so smart?"
40 PRINT "Millions of these:"
50 PRINT CHR$(21)
60 FOR I = 1 TO 256
70      PRINT CHR$(253); CHR$(254);
80 NEXT I
90 PRINT CHR$(21)
100 END
```

Check each line. Don't worry about spacing; however, if anything else is different, simply re-type the incorrect line. For example, suppose you mistakenly type in line 90 like this:

```
90 PRINT CHR$(201)
```

To correct it, simply type:

```
>90 PRINT CHR$(21) (ENTER)
>■
```

When everything is correct, you can run the program by typing:

```
>RUN (ENTER)
```

4 / Using the Keyboard

The keyboard allows entry of all the standard text and control characters. As with ordinary typewriters, use (SHIFT) to enter the upper symbol on those keys containing two symbols. For example, to enter a "!", press (SHIFT) (1).

Capitals and Lower Case (SHIFT)(0)

The A-Z keys can produce either upper or lowercase characters. There are two modes of operation: CAPS, in which the A-Z keys always produce capital letters; and ULC (upper/lowercase), in which the A-Z keys produce lowercase unless you press (SHIFT).

When you start the Computer, the keyboard is in the CAPS mode. To switch to ULC, press (SHIFT)(0). To switch back, press (SHIFT) (0) again. (SHIFT) (0) is a "toggle": each time you press it, you switch from one mode to the other.

Special Keys

Certain keys have special functions in BASIC. Rather than accepting them as keyboard data, BASIC performs the specified function.

Key	Function
(←)	Backspaces and erases the last character typed.
(→)	Tabs over to the next eight-column boundary.
(SHIFT)(←)	Starts over at the beginning of the line.
(SHIFT)(→)	Converts to 32 characters/line.
(SHIFT)(@)	Pauses program execution. Press any key to continue.
(ENTER)	Enters the line. BASIC will not interpret a line until you press (ENTER).
(CLEAR)	Cancels the current line, erases the display, converts to 64 characters/line, and positions the cursor to the upper left corner ("home").

Special Keys, continued.

(BREAK)	Interrupts the current program or operation and prepares the Computer for another keyboard command. Use to cancel a cassette or line printer operation, or to break out of a BASIC program.
(SHIFT) (↓) (*)	Activates the Print Screen function, copies the contents of the Screen to the Printer. Press (BREAK) to terminate this function and return to the immediate mode.

Other Features

Every key has a repeat feature: when you hold a key down for approximately one second, that key begins producing a stream of characters.

The keyboard includes a 12-key section for convenient numeric entry. Each of these keys is equivalent to the matching key on the standard keyboard section.

Control Codes*

* If you are unfamiliar with the concept of character codes, see the ASCII entry in the Glossary (Appendix). Also see the table of character codes in the Appendix.

You can produce 32 special control characters (ASCII Codes 0-31) from the Keyboard. For example,

Key	ASCII Name	Code
(←)	Backspace	8
(→)	Tab	9
(↓)	Line Feed	10
(ENTER)	Carriage Return	13

You are not limited to these specially labeled keys. A special two-key combination allows the regular text keys to create additional control characters. Use this procedure:

1. Hold down (SHIFT)
2. Hold down (↓)
3. While holding down (SHIFT) and (↓), press the desired character. For example: (SHIFT)(↓)(C) = "Control C" = Code # 3.

For a complete list of keyboard characters available, see the Appendix.

5 / Using the Video Display

Character Size

There are 16 lines on the display, and two character sizes: normal (64 characters per line—"cpl"), and double-size, or 32 cpl.

The Computer starts in the 64 cpl mode. To change to 32 cpl, press (SHIFT)(←) in the immediate mode or execute the BASIC statement:

```
PRINT CHR$(23)
```

To return to 64 cpl, press (CLEAR) in the command mode, or execute the BASIC statement:

```
CLS
```

Cursor

The cursor indicates the current display position. When you start BASIC, the cursor is a blinking block. You can change the cursor character and you can make it solid (non-blinking).

Memory location 16412 contains the blink/non-blink status. When it contains a zero, a blinking cursor will be used. When it contains a non-zero value, a non-blinking cursor will be used.

For example, to make a solid cursor, execute the BASIC statement:

```
POKE 16412, 1
```

To make a blinking cursor, execute the BASIC statement:

```
POKE 16412, 0
```

Memory location 16419 contains the ASCII code of the cursor character. When you start BASIC, this address contains 176. To change the cursor, use the POKE statement. For example,

```
POKE 16419, 63
```

changes the cursor to a "?", since 63 is the ASCII code for a question-mark.

You can select any ASCII code from zero to 255.

To restore the cursor to its original character, execute this BASIC statement:

POKE 16419, 176

To turn the cursor on in the execute mode, execute the statement

PRINT CHR$(14)

To turn it off, use

PRINT CHR$(15)

Scroll Protection

Display "scrolling" occurs when the Computer moves all the text up one line to make room for a new line on the bottom row of the Display. When scrolling occurs, the top line on the Display is erased from the Display.

The Model III will let you protect from scrolling up to seven lines on the top of the Display. For example, suppose you are printing a table. You can put the column headings in a scroll protect area, so they will not be lost when scrolling takes place.

Memory location 16916 controls the size of the scroll protect area. A zero in this one-byte location means no lines are protected. A one means one line (the top line) is protected. And so forth.

For example, to protect the top four lines from scrolling, execute the BASIC statement:

POKE 16916, 4

To restore the display to its original condition (no scroll-protect), execute the BASIC statement:

POKE 16916,0

If you store a value greater than seven in this address, the Computer interprets the value in modulo eight. That is, the number is divided by eight and the remainder is used.

The following program demonstrates the scroll-protect feature:

```
10 CLS: POKE 16916,3              'PROTECT TOP 3 LINES
20 PRINT "THESE TOP THREE LINES WILL NOT BE SCROLLED"
30 PRINT "BUT THE REST OF THE SCREEN WILL."
40 PRINT "-----------------------------------------------"
50 FOR I = 1 TO 100
60 PRINT "THIS LINE IS IN THE NON-PROTECTED AREA SO WILL SCROLL"
70 NEXT I
80 POKE 16916,0                   'REMOVE SCROLL PROTECTION
```

Text Characters

The Model III Display can produce the standard ASCII text characters, including the upper and lowercase alphabet.

All text characters are created on an eight-by-eight matrix for excellent definition.

The following BASIC program will display all 96 text codes and characters:

```
10 CLS
20 FOR I = 32 TO 127
30   PRINT @ (I-32) * 8, I; CHR$(I);
40 NEXT I
```

Many of these characters can be keyed in directly from the keyboard; others can only be generated by reference to their ASCII codes.

Note: The ↑ key is echoed on the display as [instead of as an up-arrow. This is because Model III produces standard ASCII characters on its display. However, if the program calls for an up-arrow, the left-bracket will serve the same purpose.

Graphics Characters

The Model III Display has 64 graphics characters, consisting of all possible on-off combinations in a two-by-three matrix:

The graphics characters are produced by codes 128 through 191. The following program will display them all:

```
10 CLS
20 FOR I = 128 TO 191
30    PRINT @ (I-128) * 8, I; CHR$(I);
40 NEXT I
```

Space Compression Characters

When you start BASIC, characters 192 through 255 are defined as space compression codes: 192 generates zero spaces: 193, one space; and so forth, up to 255, which generates 63 spaces.

These codes are useful for storing Video Display text in a minimal amount of memory. For example, the following line contains 55 characters (superior numbers indicate the number of blank spaces between letters):

21 spaces *18 spaces*

NAME ADDRESS PHONE

There are two sequences of blanks containing a total of 39 characters. By replacing the two space-sequences with two compression codes, we can save 39 – 2 = 37 characters.

When the data is displayed, the space compression codes will be "expanded" into the appropriate number of spaces.

The following BASIC program illustrates this example:

```
5  CLS
10 POKE 16526, 105                'LSB OF $INITIO ENTRY ADDRESS
20 POKE 16527, 0                  'MSB
30 X = USR(0)                     'CALL $INITIO
40 CLEAR 100
50  A$ =  "NAME"  +  CHR$(192+21) +  "ADDRESS"  +  CHR$(192+18)  +
"PHONE"
60 PRINT "THE LENGTH OF THE STRING IS"; LEN(A$)
70 PRINT "HERE IT IS:"
80 PRINT A$
```

Special Characters

The Model III also features 96 special characters. The first 32 may be displayed by POKEing the appropriate code into video RAM (addresses 15360 to 16383); the remaining 64 may be displayed via the PRINT statement.

This program will display the first 32:

```
10 CLS
20 FOR I = 0 TO 31
30      POKE 15360 + I * 16, I
40 NEXT I
50 PRINT @ 640, "";
```

The remaining 64 must first be "switched in" and then may be displayed via PRINT. Codes 192 through 255 normally function as space compression codes; however, a software switch will activate the special character set. The statement:

PRINT CHR$(21)

switches back and forth between space compression and special characters.

Another software switch selects an alternate set of special characters (Japanese Kana characters). Each time you execute the statement

PRINT CHR$(22)

the active/inactive sets are swapped.

The following program will switch in the special characters and display both sets of them.

```
5 CLS
10 POKE 16526, 105                'LSB OF $INITIO ENTRY ADDRESS
20 POKE 16527, 0                  'MSB
30 X = USR(0)                     'CALL $INITIO
40 PRINT CHR$(21)                 'SWITCH IN SPECIAL CHARACTERS
50 INPUT "PRESS <ENTER> TO SEE SPECIAL CHARACTERS"; X
60 FOR I = 192 TO 255
70  PRINT CHR$(I);
80 NEXT I
90 PRINT
100 INPUT "PRESS <ENTER> TO SWITCH TO ALTERNATE SET"; X
110 PRINT CHR$(22);               'SWITCH IN ALTERNATE SET
120 INPUT "PRESS <ENTER> TO RETURN TO NORMAL AND END"; X
130 PRINT CHR$(22); CHR$(21)
```

6 / Using the Cassette Interface

Model III's built-in cassette interface allows you to store data and programs with a cassette recorder such as Radio Shack's CTR-80A, Catalog Number 26-1206.

Connect the recorder to the Computer according to Figure 1 in this manual; for further connection instructions, refer to the cassette recorder owner's manual.

Cassette Transfer Speed

As explained previously, you select either Low or High cassette speed when you start BASIC.

If you want to load Model I Level II programs, you must select Low.

(The actual speed for Low is 500 baud, which is approximately 63 characters per second; for High, 1500 baud, or 190 characters per second. For short programs, you won't notice a three-to-one difference in loading times, due to the "overhead" required by any taped data. However, for longer programs, the difference in loading/saving times will approximate three-to-one.)

You do not have to restart BASIC to change the cassette speed. This speed is determined by the contents of memory address 16913. When this one-byte location contains zero, Low speed (500 baud) is used; when it contains any non-zero value, High speed (1500 baud) is used.

For example, to select 500 baud, execute the BASIC statement:

```
POKE 16913, 0
```

To select 1500 baud, execute the BASIC statement:

```
POKE 16913, 1
```

Loading Errors

There are three messages that may appear in the upper right of the Display during a tape input operation. They tell you that the tape operation was unsuccessful and needs to be repeated.

Message	Meaning
C*	Checksum Error during loading of a SYSTEM tape
D*	Data Error during loading of a BASIC program
BK	You pressed (BREAK) and cancelled the operation

The first two errors may be caused by an incorrect volume setting. Adjust the volume and try again. If you still have problems, recheck the cassette recorder connections. Another possible cause is dirty recorder heads. Clean the heads as explained in the cassette owner's manual. If none of this helps, the data on the tape may have been destroyed by static electricity or some other cause.

Saving a BASIC Program on Tape

When you want a long-term copy of a BASIC program (one that won't have to be typed in again), simply save it on tape with the CSAVE command.

The program should be in memory. Be sure you have selected the desired cassette transfer speed (500 or 1500 baud). In general, you should use 1500 baud, since it is faster and requires less tape.

1. Insert a blank cassette into the recorder (use Radio Shack's leaderless tape for best results).
2. Prepare the recorder to RECORD.
3. Type :

```
CSAVE "P" (ENTER)
```

The Computer will save the program on tape.

When the process is completed, the Computer will display:

```
READY
>■
```

In this example, we used "P" as the file name; you can choose any single character except a double-quote. Enclose the character in double-quotes as shown in our example.

It is a good idea to save the program at least twice, preferably on separate cassettes. That way, if one cassette is lost or erased, you have an extra copy.

When you want to load the program in later, you can specify the file name, in which case BASIC will search for that file name; or you can omit the file name, in which case BASIC will load the first program on the tape.

Loading a BASIC Program from Tape

Be sure the Computer's cassette speed matches that of the recorded program (the speed at which it was CSAVEd).

1. Prepare your recorder to PLAY the recorded cassette. Adjust the volume to the level recommended for 500 or 1500 baud. See Figure 2 on the next page.
2. Type:

   CLOAD (ENTER)

   The Computer will load the first program on the tape. While the program is loading, two asterisks will appear on the upper right of the Display. The one on the right will blink after every 64th character of data is received.

   When the program is loaded, the Computer will display the message:

   READY
   >■

3. Type:

   LIST (ENTER)

   to list the program you have just loaded (just for verification).

4. You may now run the program by typing:

   RUN (ENTER)

How to Search for a Program

If the tape contains different programs on the same side, you can make the Computer search through them until it reaches the desired program. To do this, just specify the name of the program. For example, if the program is named "P", then type in this command:

CLOAD "P" (ENTER)

While the Computer is skipping a non-matching program, it will display the file name of that program.

Note: If the program you named is not on the tape, the Computer will continue to wait for it, even after the tape has run out. Hold down the (BREAK) key until the Computer returns with the message:

```
READY
>■
```

Recorder Model	User-Generated	Pre-Recorded From Radio Shack
CTR-80, 80A	5-7	4-6

Figure 2. Recommended levels for loading programs from tape.

Loading a SYSTEM Tape

In addition to BASIC programs, you may load machine-language programs from tape. Such programs are stored in a different format on the tape; we call them SYSTEM tapes. Radio Shack sells several machine-language programs on cassette, for example, Micromusic and Editor/Assembler.

You can also create your own SYSTEM tapes, using the Editor/Assembler Package.

Before loading the tape, be sure the Computer's cassette speed matches that of the recorded program.

1. Prepare your recorder to PLAY the recorded cassette. Adjust the volume to the level recommended in Figure 2.

2. Type:
   SYSTEM (ENTER)

   The Computer will display the monitor mode prompt:

   *?

3. Type in the program's file name. For example, if the program is named EDTASM, you would type:

   EDTASM (ENTER)

   The Computer will load the program. While the program is loading, two asterisks will appear on the upper right of the Display. The one on the right will flash after every 64th character of data is received.

4. When the Computer has loaded the program, it will display another monitor prompt:

   *?

   What you do next depends on the program you have just loaded.

A. If you want to load another program, then prepare the next cassette tape and repeat Step 3.

B. If you want to return to BASIC, then press (BREAK).

C. If you want to run the machine-language program you just loaded, then type in a slash symbol "/" followed by the "entry address" and press (ENTER), or simply type in the "/" and press (ENTER). Specific instructions will be provided with the SYSTEM tape.

For example, to start the program at address 32000, type:

*?/32000 (ENTER)

To start the program at the address specified by the SYSTEM tape, type:

*?/ (ENTER)

7 / Using A Line Printer

Any Radio Shack "parallel interface" printer may be connected to the Model III. There are some differences in printer functions available, so check in the printer owner's manual for these details.

Line Printer vs Video Display Output

Output to the line printer is similar to display output; in fact, for the two major display output operations, there are two matching line printer output operations:

Video Display	Line Printer
PRINT	LPRINT
LIST	LLIST

These are described in the BASIC Language Section of this manual.

When you try to output information to the printer, the Computer will first see if a printer is connected and ready to accept the data. If it is not, the Computer will simply wait until the printer is available. During this time, you will not be able to type in instructions from the keyboard.

To regain keyboard control in this situation, hold down the (BREAK) key until the Computer displays

```
READY
>
```

Certain of the Video Display features are not available on the printer. For example:

- The graphics and special character sets cannot be output to the printer. However, your printer may have its own special characters or "graphics". Check in the owner's manual.
- The CLS and PRINT @ statements have no line-printer counterparts.

Printer Control Features

Output to a printer involves several variables:

- Maximum line width (How many print columns are there?)
- Page length (How many print lines are on a page?)
- Printer status (Is the printer connected and ready to receive data and print it?)

In this section, we will explain how to set up the Model III to control all these variables.

Setting the Maximum Line Length

In Model III BASIC, you can preset the maximum line length. If a line exceeds the preset length, the Computer will automatically insert an end of line (carriage return) so that the rest of the line will be output on a new line. The following paragraphs explain why you may want to do this.

One important difference between display output and printer output is the maximum line length. (A "line" is a stream of data characters terminated by a carriage return (ENTER).)

The Model III Display has a maximum line length of 64 characters. If you PRINT a line longer than this, the Computer simply "wraps around" to the beginning of the next line.

Printers have a maximum line length, too, but this length differs for various models. The response to an overflow (longer than maximum-length) line also varies. Some models wrap around to the next line automatically. Others may lose the extra data, and may begin abnormal operation when the line is too long.

Another consideration is paper width. Suppose your paper is only wide enough to hold 80 characters—but the printer will accept lines of up to 132 characters. In this case, if you send a line longer than 80 characters, the printer will print part of the information past the edge of the paper.

How to Set the Line Length

Memory address 16427 contains a value equal to the maximum line length less two. For example, to set the maximum line length to 64, execute the BASIC statement:

```
POKE 16427, 62
```

Since the Display is 64 characters per line (cpl), this setting will make line printer output match Video Display output.

When address 16427 contains a value of 255, the maximum line length feature is disabled. No matter how long the line is, the Computer will not insert carriage returns in it. Remember, though, some printers automatically do this when the line exceeds a specified length.

When you start BASIC, address 16427 contains a value of 255, so the maximum line length function is disabled.

Page Controls

In many printer applications, you want to control the number of lines that are printed on a page. For example, in printing forms or reports, when a given number of lines have been printed, you want to advance the paper to the top of the next page.

Model III BASIC has several features to help you do this. It keeps track of the following information:

Data	Memory Address
Page size: number of lines per page plus one. Initialized to 67 = 66 + 1.	16424
Line count: number of lines (carriage returns) already printed plus one. Initialized to one.	16425

Most printers output six lines per inch; therefore standard 11″ paper allows 66 lines, which matches BASIC's initialization value.

To change the maximum lines/page setting, store the desired number of lines plus one in 16424. For example, if your paper contains 88 lines per page, then execute this BASIC statement:

```
POKE 16424, 89
```

When you start the Computer, position the paper to the top of the page (''top of form''). That way BASIC's initial page information is correct. Each time BASIC outputs a line (i.e., a carriage return), the line count is incremented.

Note: If your printer's maximum line-length is shorter than BASIC's maximum line length, the printer will insert carriage returns that BASIC isn't allowing for. Therefore BASIC's line count will not be accurate.

To prevent this from happening, make sure BASIC's maximum line length (stored in address 16427) is no greater than that of your printer. You can find your printer's maximum line length in the printer owner's manual.

To do an automatic top of form (advancing the paper to the top of the next page), print the ASCII "Form Feed" code, decimal 12. For example, execute the BASIC statement:

```
LPRINT CHR$(12)
```

The paper will advance by the following amount:

Top of Form = Max. lines/page — Lines already printed

Each time you print a form feed, CHR$(12), BASIC resets the line count automatically.

Sometimes you may want to reset the line count, for example, after manually advancing the paper to the top of form. To do this, store a one in 16425:

```
POKE 16425, 1
```

Checking the Printer Status

Unlike the Video Display, the printer is not always available. It may be disconnected, off-line, out-of-paper, and so forth. In such cases, when you try printer output, the Computer will wait until the printer becomes available. It will appear to be "locked up". To regain keyboard control (and cancel the printer operation), press (BREAK).

Suppose you have a program which uses printer output. If a printer is not available, you don't want the Computer to stop and wait for it to become available. Instead, you may want to print a message like "PRINTER UNAVAILABLE" and stop.

To accomplish this, you need to check the printer status. The status is stored in address 14312. AND this value with 240. The result should equal 48. If it doesn't, that means the Printer is unavailable for some reason, and printer output is not possible. For example, your program could execute these statements:

```
100 ST% = PEEK(14312) AND 240
120 IF ST% < > 48 THEN PRINT "PRINTER UNAVAILABLE.": STOP
130 PRINT "PRINTER IS AVAILABLE"
```

Print Screen Function

Model III has a very handy feature to give you a "snapshot" of whatever is on the Display. It will work whenever the Computer is scanning the keyboard (BASIC's Immediate, Execute, Edit and System Modes). It does not work during cassette, printer or serial I/O.

When you want to copy the Display contents to the printer, simply press:

(SHIFT) (↓) (*)

together. The Computer will stop what it's doing and print the screen.

The Computer will print the entire display, blanks and all. If you are only interested in printing the top portion of the display, press (BREAK) when those lines have been printed.

If a printer is not available, the Computer will wait until it becomes available or until you press (BREAK).

If the Display contains special characters or graphics characters, they will be displayed as periods.

Note: You can also activate the Print-Screen function via the BASIC USR function. See $PRSCRN in the Technical Information chapter.

8 / Using the RS-232-C Interface

What is an Interface?

It's a generalized means of communication between your TRS-80 and some external device, providing the necessary conventions regarding data-identification, transmission rates, send-receive sequences, error-checking techniques, etc. However, an Interface does **not** provide the programming necessary to **use** any particular TRS-80/ external device system.

For example, having the Interface installed does **not** automatically enable you to send BASIC programs from one TRS-80 to another; to output to a line printer via the Interface; etc. Such applications require ''driver programs'' which must be custom-designed for the equipment you intend to use.

The Radio Shack RS-232-C Interface is designed to meet the EIA standards. However, we cannot guarantee that it will work with all so-called ''RS-232-C compatible'' devices. Nor do we commit ourselves to provide engineering and programming support for such applications, or other special custom-use situations.

We do, however, guarantee that our Interface will function correctly with all our own RS-232-C equipment.

The term RS-232-C refers to a specific EIA (Electronics Industries Association) standard which defines a widely accepted method for interfacing data terminal equipment with data communications equipment. The RS-232-C Interface is by far the most universally used standard for interfacing data processing equipment. Most video terminals, modems, card readers, line printers, mini-microcomputers, etc., utilize the RS-232-C standard for data interchange between devices.

Adding the RS-232-C to your Model III TRS-80 opens up a whole new world of compatibility. The Computer can then be programmed to communicate with a serial printer, telephone modem, serial display terminal — almost any RS 232-C device.

Note: The following information applies only if your Model III TRS-80 is equipped with the RS-232-C Interface.

Using the Model III as a Terminal

Probably the most common use of the RS-232-C interface will be to allow the Model III to act like a "terminal" to another "host" computer. In this application, whatever you type on the keyboard is sent via RS-232-C to the other host computer, and whatever the host computer sends to you is displayed on your screen.

Before going into the details of RS-232-C operation, we'll show you a BASIC program that sets up a **simplified** terminal operation.

1. Make sure the RS-232-C characteristics are set to match those of the host computer. If they are not, then change them, as explained later in this chapter.

Note: For this BASIC Program, you must use a baud rate of 110. An equivalent Z-80 program could use any baud rate.

2. Connect the Model III to the host computer via the RS-232-C. You will need a telephone interface (modem) or other means of communication.

3. Type in and run the following BASIC program (you do not need to type in the comments (material that starts with a single quote). The program displays characters received via the RS-232-C, and sends characters you type in. It is for demonstration only, and is not meant to function as a practical terminal. Notice there are *no spaces* between the " " in line 160.

```
5 DEFINT A-Z                    'INTEGER VARIABLE FOR SPEED
10 POKE 16890, 0                'DON'T WAIT FOR SERIAL I/O
15 POKE 16888,(2*16)+2          'TX/RCV AT BAUD RATE 110
17 U1 = 16526                   'LSB OF USR CALL ADDRESS
18 U2 = 16527                   'MSB OF USR CALL ADDRESS
20 POKE U1, 90                  'SET UP USR CALL, LSB
30 POKE U2, 0                   '      MSB
40 X = USR(0)                   'CALL $RSINIT
50 RCV = 80                     'LSB OF $RSRCV
60 TX = 85                      'LSB OF $RSTX
70 CI = 16872                   'CHARACTER INPUT BUFFER
80 CO = 16880                   'CHARACTER OUTPUT BUFFER
90 '  CHECK FOR SERIAL INPUT
100 POKE U1, RCV                'SET UP USR CALL TO $RSRCV
110 X = USR(0)                  'CALL $RSRCV
120 C$ = CHR$(PEEK(CI))         'LOOK AT INPUT BUFFER
130 PRINT C$;                   'IF C = 0, NOTHING HAPPENS
140 '  CHECK FOR KEYBOARD INPUT
150 C$ = INKEY$
160 IF C$ = "" THEN 100         'NO KEY, SO GO CHECK SERIAL
165 PRINT C$;                   'DELETE THIS LINE IF HOST PROGRAM
166                             'HAS AN ECHO FEATURE
170 POKE CO, ASC(C$)            'PUT CHAR. INTO OUTPUT BUFFER
180 POKE U1, TX                 'SET UP USR CALL TO $RSTX
190 X = USR(0)                  'CALL $RSTX
200 GOTO 100                    'GO CHECK SERIAL INPUT
```

Programming the RS-232-C Interface

In this section, we will treat the RS-232-C just like any other input/output device, and will explain how your BASIC program can use it. In Technical Information, we explain how to use it in a machine-language ("Z-80") program.

For details about the RS-232-C signal conventions and theory of operation, see the Appendix.

Selecting the RS-232-C Characteristics

Before using the RS-232-C interface to communicate with another device, you must be sure your RS-232-C is set up to match the requirements of the other device.

So start by getting the following information about the other device. In the right column, we list typical values used.

Characteristic	Typical Values Used
Baud Rate	110, 150, 300, 600, 1200, 2400, 4800, 9600
Word Length (bits)	5, 6, 7, 8
Parity	Even, Odd, None
Stop Bits	1, 2

When you start the Computer, the RS-232-C is initialized to the following "default characteristics":

Baud Rate	300
Word Length (bits)	8
Parity	None
Stop Bits	1

In addition, the RS-232-C is initialized to wait for completion of character I/O before returning. That is, if you attempt to receive a character, the Computer will wait until a character is received; it will never return to you without a character. Similarly, if you attempt to send a character, the Computer will wait until the receiving device is able to accept the character.

To regain control of the Computer during a wait, hold (BREAK) until READY returns.

I/O to the RS-232-C Interface

If the default settings are correct, you are ready to begin serial I/O. To change any of the settings, you need to re-initialize the RS-232-C interface. See "To Change the RS-232-C Characteristics".

There are two ROM subroutines for serial I/O (both were used in the simple terminal program):

$RSTX	Send a character
$RSRCV	Receive a character

Both subroutines are simple to use from BASIC via the USR function.

To Send a Character

1. The Computer should be connected to the serial device.
2. Define a USR call to $RSTX (address 85) by executing these BASIC statements:

```
POKE 16526, 85
POKE 16527, 0
```

3. Send the character by storing the ASCII code in memory location 16880. Suppose A$ contains the character. Then execute this statement:

```
POKE 16880, ASC(A$)
```

4. Make the USR call with a dummy argument:

```
X = USR(0)
```

   If the Computer is using the Don't Wait procedure, then control will return to BASIC even if the character was not sent. If the Computer is using the Wait procedure, control will return to BASIC after the character is sent.

5. Repeat steps 3 and 4 until all the data has been sent.

To Receive a Character

1. The Computer should be connected to the serial device.
2. Define a USR call to $RSRCV (address 50) by executing these BASIC statements:

```
POKE 16526, 50
POKE 16527, 0
```

3. Get the character by making the USR call with a dummy argument. For example:

```
X = USR(0)
```

   Upon return from the subroutine, USR returns the ASCII code of the character received in memory location 16872. A zero indicates no value was received.

If the Computer is using the Don't Wait procedure, then control will return to BASIC even if no character was received. If the Computer is using the Wait procedure, control will return to BASIC after a character is received. Press (BREAK) to interrupt a WAIT and regain keyboard control of the Computer.

4. To make this character available to BASIC, execute a BASIC statement like:

   A$ = CHR$(PEEK(16872))

   which stores the string value in A$. Remember, if A$ = CHR$(0), then no character was received.

5. Repeat Steps 3 and 4 until you are through receiving data.

To Change the RS-232-C Characteristics

If the TRS-80's default characteristics do not match the requirements of the other device, you can change some or all of them by using (''calling'') an initialization subroutine that is stored in ROM.

Before calling $RSINIT, you must store the desired characteristics in certain RAM locations:

Address	Contents
16888	Transmit/Receive Baud Rate Code
16889	Parity/Word Length/Stop Bit Code
16890	Wait/Don't-Wait Switch

Transmit/Receive Baud Rate Code

The TRS-80 RS-232-C allows you to receive and transmit at different rates. For most applications, the rates will need to be the same.

Instead of storing the actual baud rate, you store a code for the value, taken from the table below. You select the appropriate codes for send and receive rates, and then ''pack'' them into memory address 16888 as follows:

*Send/Receive Code = (Send Code * 16) + Receive Code*

For example, suppose we want to send and receive at 110 baud. Using the table on the next page, we find that the code for 110 baud is 2. So:

*Send/Receive Code = (2 * 16) + 2 = 34*

In technical terms, we are storing the send-rate code in the most significant four bits (''nibble'') of 16888, and the receive-code in the least significant nibble.

Baud-Rate Codes

Desired Baud Rate	Error (%)	Baud Rate Code
50	0	0
75	0	1
110	0	2
134.5	0.016	3
150	0	4
300	0	5
600	0	6
1200	0	7
1800	0	8
2000	0.253	9
2400	0	10
3600	0	11
4800	0	12
7200	0	13
9600	0	14
19200	3.125	15

Parity/Word Length/Stop-Bit Code

You pack all of this information into one byte, using the following formula:

$$Code = (Parityselect * 128) + (Word * 32) + (Stop * 16) + (Parityonoff * 8) + (Transmit * 4) + (DTR * 2) + RTS$$

where:

Parityselect = 0 for odd parity
= 1 for even parity

Word = 0 for 5-bit words
= 1 for 6-bit words
= 2 for 7-bit words
= 3 for 8-bit words

Stop = 0 for 1 stop-bit
= 1 for 2 stop-bit

Parityonoff = 0 to enable parity
= 1 to disable parity

Transmit	= 0 to disable the transmitter = 1 to enable the transmitter
DTR	= 0 to set Data Terminal Ready signal low = 1 to set Data Terminal Ready signal high
RTS	= 0 to set Request to Send signal low = 1 to set Request to Send signal high

For example, to select 7-bit words, even parity, two stop-bits, transmit-enable, DTR high and RTS high, calculate the code this way:

$$Code = (1 * 128) + (2 * 32) + (1 * 16) + (0 * 8) + (1 * 4) + (1 * 2) + (1 * 1) = 215$$

For additional information on how to determine the appropriate code characteristics, read $RSINIT in the Technical Information Chapter and see Appendix I.

Wait/Don't-Wait Switch

The TRS-80 lets you choose either Wait or Don't-Wait serial I/O.

When you select Wait I/O, the TRS-80 will not return from a serial I/O call until the operation is successful (i.e., a character is transmitted or received). Pressing (BREAK) will return control to your program.

When you select Don't-Wait I/O, the TRS-80 will return from a serial I/O call even if the operation was not successful (i.e., no character was transmitted or received).

The contents of memory location 16890 determines which procedure is used:

Contents of 16890	Procedure Used
Zero	Don't-Wait
Non-Zero	Wait

Calling $RSINIT from BASIC

Store (POKE) the desired values into the RS-232-C control addresses (16888-16890). If any of the default characteristics are already correct, leave those addresses unchanged.

If you need to change the parity/word length/stop-bit code, see $RSINIT in the Technical Information chapter. Once you have calculated the desired codes for baud rate, parity/word length/stop-bits and Wait/Don't-Wait, you are ready to call $RSINIT.

Execute the following BASIC statements to define a USR call to $RSINIT:

```
POKE 16526, 90
POKE 16527, 0
X = USR(0)
```

When the last statement has been executed, the RS-232-C is initialized.

9 / Routing Input/Output

Model III lets you route I/O from one device to another. This gives your programs more versatility.

For example, suppose you have a program that outputs to the Video Display. Now suppose you want all display output to go to the printer. You can accomplish this without changing the program at all, using the route capability. The source device (in our example, the display) will then be logically equivalent to the destination device (printer) until you re-initialize the I/O drivers with $INITIO (described later).

Here are the devices that may be routed:

Device	System Abbreviation
Keyboard	KI
Display	DO
Printer	PR
RS-232-C	
Send	RO
Receive	RI

To Route from One Device to Another

Note: To actually try out the next four steps, you must have printer connected to your Computer. If not, just read through the example.

1. Store the Source Device Abbreviation in memory locations 16930-16931. For example, to store DO (display) as the source device, execute the BASIC statements:

   ```
   POKE 16930, ASC("D")
   POKE 16931, ASC("O")
   ```

2. Store the Destination Device Abbreviation in memory locations 16928-16929. For example, to store PR (printer) as the destination device, execute the BASIC statements:

   ```
   POKE 16928, ASC("P")
   POKE 16929, ASC("R")
   ```

3. Set up a USR call to $ROUTE (address 108). For example, execute the BASIC statements;

   ```
   POKE 16526, 108
   POKE 16527, 0
   ```

4. Make a USR call to $ROUTE with a dummy argument. For example, execute the BASIC statements:

   ```
   X = USR(0)
   ```

   Upon completion of Step 4, the route is completed. Now everything you send to the display will be sent to the printer instead.

Routing Multiple Devices

You can change two or more of the I/O routes. To do this, you perform the routing Steps 1 through 4 once for each change you wish to make. However, to get the desired result, you must do the changes **in the correct order**! If you use one device as the **source** of a route, you should not later on use the same device as a destination. Here's why:

After you route device A to device B, device A is now logically equivalent to device B. Therefore:

(1) Route A to B
(2) Route C to A

Does not allow C to output to device A. Output to C will actually transfer to B, just as if you had executed these steps:

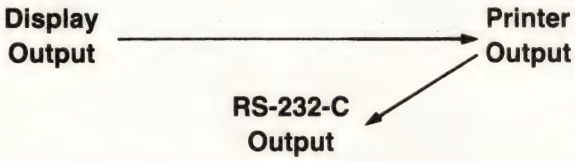
(1) Route A to B
(2) Route C to B

On the other hand:

(1) Route C to A
(2) Route A to B

Does allow device C to output to device A and device A to output to device B.

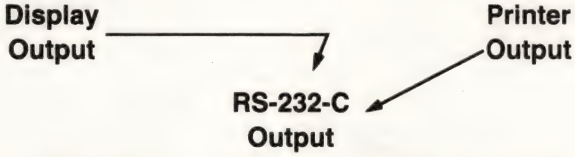
For example, suppose you want to route display output to the printer, and printer output to the RS-232-C. Here's a diagram of what you want to accomplish:



Display output goes to the Printer, and Printer output goes to the RS-232-C. All other I/O routes are unchanged. Note that Display output does **not** get carried forward from the Printer to the RS-232-C. To accomplish the routing pictured above, use this sequence:

1. Route DO to PR
2. Route PR to RO

If you mistakenly do the steps in reverse order, you will get this result:



In this case, Display output is "carried forward" from the printer to the RS-232-C. It does not output to the printer.

10 / Real-Time Clock

The Model III contains a real-time clock. It is always running, except during cassette and disk I/O and during certain other operations.

The clock keeps the following information in memory:

Abbrev.	Range of Values		Memory Location
MO	Month	01 - 12	16924
DA	Day	01 - 31	16923
YR	Year	00 - 99	16922
HR	Hour	00 - 23	16921
MN	Min.	00 - 59	16920
SS	Sec.	00 - 59	16919

The clock includes the logic for 28, 30 and 31-day months. It does not recognize leap years.

When you start the Computer, the clock is set to all zeroes:

00/00/00 00:00:00

To Set the Clock

Simply store the appropriate data in the memory addresses given above. You may do this by running the following program:

```
10 DEFINT A-Z
20 DIM TM(5)
30 CL = 16924
40 PRINT "INPUT 6 VALUES: MO, DA, YR, HR, MN, SS"
50 INPUT TM(0), TM(1), TM(2), TM(3), TM(4), TM(5)
60 FOR I = 0 TO 5
70        POKE CL - I, TM(I)
80 NEXT I
90 PRINT "CLOCK IS SET"
100 END
```

To Read the Clock

The Model III includes a built-in BASIC function, TIME$, to get the time in a 17-byte string. For example, execute the BASIC statement:

```
PRINT TIME$
```

To display the time.

To Display the Clock in Real-Time

You can turn on a continuously updated clock display. The current time (not the date) will be displayed in columns 57 - 64, regardless of what mode the BASIC is in: Immediate, Execute, Edit, or System. **As long as the clock is running,** it will be updated on the display.

To enable the clock display, call the ROM subroutine $CLKON at address 664. To disable it, call the ROM subroutine $CLKOFF at 673.

The following BASIC program shows how to turn the display on and off. Each time you want to switch it on or off, run the program.

Note: To calculate the most significant and least significant bytes of a decimal number, use this formula:

MSB = *integer portion of (number/256)*
LSB = *number - (MSB * 256)*

For example, decimal address 661 can be broken down this way:

MSB = *integer portion of (661/256) = 2*
LSB = *661 - (2 * 256) = 152*

Sample Program

```
5 CLS
10 DEFINT A-Z
20 EN = 152: DI = 161          'LSB OF $CLKON/$CLKOFF
30 PRINT "<E> NABLE CLOCK DISPLAY"
40 PRINT "<D> ISABLE CLOCK DISPLAY"
50 INPUT A$
60 IF A$ = "E" THEN SW = EN: GOTO 100
70 IF A$ = "D" THEN SW = DI: GOTO 100
80 GOTO 30
100  POKE 16526, SW            'SET UP USR CALL
110 POKE 16527, 2              'MSB IS SAME FOR BOTH CALLS
120 X = USR(0)                 'CALL USR SUBROUTINE
130 END
```

For further information about the real-time clock, see $CLKON and $CLKOFF in the Technical Information chapter.

11 / Input/Output Initialization

Whenever you start or reset the Computer, the input/output routines (''I/O drivers'') are initialized to their default values (as explained in the following chapters). For example, the Video Display is initialized to have a blinking cursor.

As described in the previous chapters, there are ways for you to alter these default characteristics via a BASIC or Z-80 program. Because of this feature, it is important to have a means of resetting the I/O drivers to their default conditions.

Model III has a ROM subroutine to re-initialize all I/O drivers to their default values. We call it $INITIO.

The following BASIC program shows how to use $INITIO.

```
10 POKE 16526, 105          'LSB OF $INITIO ENTRY ADDRESS
20 POKE 16527, 0            'MSB
30 X = USR(0)               'CALL $INITIO
```

Run this program whenever you want to restore the I/O drivers to their initial characteristics.

12/Technical Information

This section is intended for Z-80 programmers and BASIC programmers who are familiar with binary and hexadecimal arithmetic and hardware concepts like bit and byte. Its purpose is to allow you to take full advantage of the Model III TRS-80.

If you want to understand and use the system on this level, but do not have the background, we suggest you read:

TRS-80 Assembly Language Programming
by William Barden, Jr.
Radio Shack Catalog Number 62-2006

This one book will get you off to a good start. It was written for the Model I TRS-80, but almost all of it applies to the Model III as well.

To Protect High RAM

In many applications, you will want to interface a BASIC program and a Z-80 routine. In such cases, you need to protect enough high RAM to accommodate your Z-80 routine. Otherwise, BASIC will use all RAM available for storage and execution of the BASIC program.

During the start-up dialog, you have the option of protecting high RAM via the Memory Size Question. If you simply press (ENTER) to this question, BASIC will use all available RAM.

To protect RAM, type in the "limit address" in decimal form, and then press (ENTER). The limit address is the highest memory address you want BASIC to use. Addresses above this value will not be affected by BASIC.

For example, if you type: "32667 (ENTER)", BASIC will not use any memory above 32667. It **will** use 32667 and all lower-numbered memory locations.

ROM Subroutines

The Model III BASIC ROM contains many subroutines that can be called by a Z-80 program; many of these can be called by a BASIC program via the USR function. Each subroutine will be described in the format given below.

Important Note

Some of these ROM addresses or calling procedures may change in later releases of the Model III ROM. We suggest you design your programs to minimize the difficulty of adjusting to these possible changes. (Use EQUates for all ROM calls; modularize all uses of ROM routines; etcetera.)

1. $NAME — Entry address

2. Function Summary

3. Description of function

4. Entry Conditions

5. Exit Conditions

6. Sample Program

Notes:

1. The subroutine name is only for convenient reference. It is not recognized by the Computer. The $- prefix reminds you that it is a convenience name only.

The entry address is given in decimal/hexadecimal form. (The hexadecimal address will be given in this form: X'0000'.) This is the address you use in a Z-80 CALL. BASIC programmers store this address in the USR definition address (16526-16527).

4, 5. Entry and exit conditions are given for Z-80 programs. If a Z-80 register is not mentioned here, then you can assume it is unchanged by the subroutine.

6. Sample Program fragments are given in Z-80 Assembly Language and, where appropriate, in BASIC.

Here are the subroutines, arranged according to function. In the following pages, they are arranged alphabetically.

System Control

$CLKON	Clock-display on
$CLKOFF	Clock-display off
$DATE	Get today's date
$DELAY	Delay for a specified interval
$INITIO	Initialize all I/O drivers
$READY	Jump to Model III "Ready"
$RESET	Reset Computer
$ROUTE	Change I/O device routing
$SETCAS	Prompt user to set cassette baud rate
$TIME	Get the time

Cassette I/O

$CSHIN	Cassette on, search for leader and sync byte
$CSIN	Input a byte
$CSOFF	Turn off cassette drive
$CSHWR	Cassette on, Write leader and sync byte
$CSOUT	Write a byte to cassette

Keyboard Input

$KBCHAR	Get a character if available
$KBWAIT	Wait for a character
$KBLINE	Wait for a line
$KBBRK	Check for (BREAK) key only

Printer Output

$PRCHAR	Print a character
$PRSCN	Print entire screen contents

RS-232-C I/O

$RSINIT	Initialization
$RSRCV	Receive a character
$RSTX	Send a character

Video Display Output

$VDCHAR	Display a character
$VDCLS	Clear the screen
$VDLINE	Display a line

```
                    00001
                    00002 ;         MODEL III ROM CALLS - DEMONSTRATION PROGRAM
                    00003 ;
                    00004 ;         CREATED 07/07/80
                    00005 ;         UPDATED 07/08/80
                    00006 ;
                    00007 ;         TO DEMONSTRATE, JUMP TO THE APPROPRIATE ENTRY
                    00008 ;         POINT. EACH DEMO ENDS WITH A JUMP TO BASIC 'READY'
                    00009 ;
                    00010 ;
0000                00011 RESET   EQU     0000H
002B                00012 KBCHAR  EQU     002BH
0033                00013 VDCHAR  EQU     0033H
003B                00014 PRCHAR  EQU     003BH
0040                00015 KBLINE  EQU     0040H
0049                00016 KBWAIT  EQU     0049H
0050                00017 RSRCV   EQU     0050H
0055                00018 RSTX    EQU     0055H
005A                00019 RSINIT  EQU     005AH
0060                00020 DELAY   EQU     0060H
0069                00021 INITIO  EQU     0069H
006C                00022 ROUTE   EQU     006CH
01C9                00023 VDCLS   EQU     01C9H
01D9                00024 PRSCN   EQU     01D9H
01F8                00025 CSOFF   EQU     01F8H
021B                00026 VDLINE  EQU     021BH
0235                00027 CSIN    EQU     0235H
0264                00028 CSOUT   EQU     0264H
0287                00029 CSHWR   EQU     0287H
028D                00030 KBBRK   EQU     028DH
0296                00031 CSHIN   EQU     0296H
0298                00032 CLKON   EQU     0298H
02A1                00033 CLKOFF  EQU     02A1H
3042                00034 SETCAS  EQU     3042H
1A19                00035 READY   EQU     1A19H
3033                00036 DATE    EQU     3033H
3036                00037 TIME    EQU     3036H
37E8                00038 PRSTAT  EQU     37E8H
                    00039 ;-----------------------------------------------------------------
8000                00040         ORG     8000H
                    00041 ;
```

Note: This Z-80 assembly language listing is continued under the ROM call entries for **Sample Z-80 Programming.**

$CLKOFF — 673/X'02A1'

Disable the Clock Display

Entry Conditions

None

Exit Conditions

A is altered. All other registers are unchanged.

Sample Z-80 Programming

```
                        00042 ; TURN OFF CLOCK
8000    CDA102          00043          CALL     CLKOFF
8003    C3191A          00044          JP       READY
```

Sample BASIC Programming

```
100 POKE 16526,161: POKE 16527,2          'LSB/MSB
110 X = USR(0)                            'DUMMY ARGUMENT
```

$CLKON — 664/X'0298'

Enable the Clock Display

Entry Conditions

None

Exit Conditions

A is altered. All other registers are unchanged.

Sample Z-80 Programming

```
                        00045 ; TURN ON CLOCK
8006    CD9802          00046          CALL     CLKON
8009    C3191A          00047          JP       READY
```

Sample BASIC Programming

```
100 POKE 16526,152: POKE 16527,2          'LSB/MSB
110 X = USR(0)                            'DUMMY ARGUMENT
```

$CSHIN — 662/X'0296'

Search for Cassette Header and Sync Byte

Each cassette "record" begins with a header consisting of a leader sequence and synchronization byte. $CSHIN turns on the cassette drive and begins searching for this header information. The subroutine returns to the calling program after the sync-byte has been read.

Entry Conditions

None

Exit Conditions

A is altered. All other registers are unchanged.

Sample Z-80 Programming

The following program reads the tape created by the $CSHWR sample program.

```
                        00048 ; READ A MESSAGE FROM TAPE & STOP ON CAR-RET'N
800C    CDC901          00049          CALL    VDCLS            CLEAR SCREEN
800F    3E0D            00050          LD      A,0DH
8011    CD3300          00051          CALL    VDCHAR           SKIP A LINE
8014    CD4230          00052          CALL    SETCAS           LET USER SELECT BAUD RATE
8017    213B80          00053          LD      HL,MSG0          (HL)=CASSETTE PROMPT
801A    CD1B02          00054          CALL    VDLINE
801D    CD4900          00055          CALL    KBWAIT           WAIT FOR ANY KEY
8020    216280          00056          LD      HL,TXT           (HL)=256-BYTE BUFFER
8023    CD9602          00057          CALL    CSHIN            FIND START OF RECORD
8026    CD3502          00058 LOOP     CALL    CSIN             INPUT A BYTE
8029    77              00059          LD      (HL),A           STORE IT
802A    23              00060          INC     HL               POINT TO NXT LOC.
802B    FE0D            00061          CP      0DH              WAS LAST BYTE=CAR-RET'N?
802D    20F7            00062          JR      NZ,LOOP          IF NO, GET NXT BYTE
802F    CDF801          00063          CALL    CSOFF            IF YES, TURN OFF CASSETTE
8032    216280          00064          LD      HL,TXT           DISPLAY THE MESSAGE
8035    CD1B02          00065          CALL    VDLINE
8038    C3191A          00066          JP      READY            AND QUIT
803B    50              00067 MSG0     DEFM    'PREPARE TAPE TO PLAY AND PRESS ANY KEY'
8061    0D              00068          DEFB    0DH
8062                    00069 TXT      DEFS    256              STORAGE FOR TAPED MESSAGEE
```

$CSIN — 565/X'0235'

Input a Byte

After completion of $CSHIN, use $CSIN to begin inputting data, one byte at a time.

Note: You must call $CSIN often enough to keep up with the baud rate (either 500 or 1500 baud).

Entry Conditions

None

Exit Conditions

A = Data byte

Sample Z-80 Programming

See $CSHIN.

$CSHWR — 647/X'0287'

Write Leader and Sync Byte

Each cassette "record" begins with a header consisting of a leader sequence and a synchronization byte. $CSHWR turns on the cassette and writes out this header.

Entry Conditions

None

Exit Conditions

A is altered.

Sample Z-80 Programming

```
                        00070 ;INPUT A KEYBOARD MESSAGE AND WRITE IT TO CASSETTE
8162    CDC901          00071          CALL    VDCLS
8165    3E0D            00072 LOOP1    LD      A,0DH            CARRIAGE RETURN
8167    CD3300          00073          CALL    VDCHAR           SKIP TO NEXT DISPLAY LINE
816A    21A081          00074          LD      HL,MSG1          PROMPT MESSSAGE
816D    CD1B02          00075          CALL    VDLINE           DISPLAY IT
8170    21EA81          00076          LD      HL,TXT1          256-BYTE BUFFER
8173    06FF            00077          LD      B,255            MAX OF 255 CHARACTERS
8175    CD4000          00078          CALL    KBLINE           GET A LINE FROM KB
8178    38EB            00079          JR      C,LOOP1          LOOP IF <BREAK> WAS PRESSED
817A    3E0D            00080          LD      A,0DH
817C    CD3300          00081          CALL    VDCHAR           SKIP A LINE
817F    CD4230          00082          CALL    SETCAS           LET USER SELECT BAUD RATE
8182    21B381          00083          LD      HL,MSG2          CASSETTE PROMPT
8185    CD1B02          00084          CALL    VDLINE
8188    CD4900          00085          CALL    KBWAIT           WAIT UNTIL A KEY IS PRESSED
818B    CD8702          00086          CALL    CSHWR            WRITE CASSETTE HEADER
818E    21EA81          00087          LD      HL,TXT1          (HL)=MESSAGE
8191    7E              00088 LOOP2    LD      A,(HL)           A=ASCII BYTE
8192    23              00089          INC     HL               POINT TO NEXT BYTE
8193    CD6402          00090          CALL    CSOUT            WRITE LAST BYTE TO TAPE
8196    FE0D            00091          CP      0DH              WAS IT A CARRIAGE RETURN?
8198    20F7            00092          JR      NZ,LOOP2         IF NO, THEN GET NEXT BYTE
819A    CDF801          00093          CALL    CSOFF            IF YES, TURN OFF CASSETTE
819D    C3191A          00094          JP      READY
81A0    54              00095 MSG1     DEFM    'TYPE IN A MESSSAGE'
81B2    0D              00096          DEFB    0DH
81B3    4D              00097 MSG2     DEFM    'MESSSAGE STORED. PRESS ANY KEY WHEN READY TO RECORD...
81E9    0D              00098          DEFB    0DH              END OF LINE
81EA                    00099 TXT1     DEFS    256
```

For a program to read the tape in, see $CSHIN.

$CSOFF — 504/X'01F8'

Turn Off Cassette

After writing data to cassette, call this subroutine to turn off the cassette drive.

Entry Conditions

None

Exit Conditions

None

Sample Z-80 Programming

See $CSHWR.

$CSOUT — 612/X'0264'

Output a Byte to Cassette

After writing the header with $CSHWR, use $CSOUT to write the data, one byte at a time.

Note: You must call $CSOUT often enough to keep up with the baud rate (either 500 or 1500 baud).

Entry Conditions

A = Data byte.

Exit Conditions

None

Sample Z-80 Programming

See $CSHWR.

$DATE — 12339/X'3033'

Get Today's Date

Entry Conditions

(HL) = Eight-byte output buffer

Exit Conditions

(HL) = Date in this format:
MO/DA/YR

All other registers are altered.

Sample Z-80 Programming

```
                        00100 ; GET TODAY'S DATE & TIME
82EA    210883          00101          LD      HL,TXT2          8-BYTE BUFFER
82ED    CD3330          00102          CALL    DATE
82F0    21FF82          00103          LD      HL,TXT3          8-BYTE BUFFER
82F3    CD3630          00104          CALL    TIME
82F6    21FF82          00105          LD      HL,TXT3          (HL)=TIME/DATE MSG.
82F9    CD1B02          00106          CALL    VDLINE           DISPLAY TIME/DATE
82FC    C3191A          00107          JP      READY
82FF                    00108 TXT3     DEFS    8                TIME GOES HERE
8307    20              00109          DEFB    20H              ASCII SPACE
8308                    00110 TXT2     DEFS    8                DATE GOES HERE
8310    0D              00111          DEFB    0DH              END OF LINE
```

$DELAY — 96/X'0060'

Delay for a Specified Interval

This is a general-purpose routine to be used whenever you want to pause before continuing with a program.

Entry Conditions

BC = Delay multiplier. Actual delay will be:

2.46 + (14.8 * BC) microseconds

When BC = 0000, 65536 is used. This is the maximum delay (about one second).

Exit Conditions

BC and A are altered.

Sample Z-80 Programming

```
                        00112 ;SHOW ALL DISPLAY CHARACTERS, WITH DELAY AFTER EACH
3E20                    00113 CENTER  EQU     3E20H          ROW 8, COLUMN 32 OF VIDEO
8311    CD6900          00114         CALL    INITIO         RESTORE ALL I/O DRIVERS
8314    CDC901          00115         CALL    VDCLS          FIRST CLEAR SCREEN
8317    3E00            00116         LD      A,0H
8319    01FF7F          00117         LD      BC,7FFFH       SET 1/2-SEC DELAY FACTOR
831C    32203E          00118 LOOP3   LD      (CENTER),A     WRITE CHARACTER TO VIDEO
831F    F5              00119         PUSH    AF             SAVE LAST CHAR. CODE
8320    C5              00120         PUSH    BC             AND DELAY FACTOR
8321    CD6000          00121         CALL    DELAY
8324    C1              00122         POP     BC
8325    F1              00123         POP     AF
8326    3C              00124         INC     A              NEXT CHAR CODE
8327    20F3            00125         JR      NZ,LOOP3       IF NOT ZERO, DISPLAY IT
8329    C3191A          00126         JP      READY          ELSE END
```

$INITIO — 105/X'0069'

Initialize All I/O Drivers

Call $INITIO to restore all I/O drivers to their initial default conditions, including I/O routes.

Entry Conditions

None

Exit Conditions

All registers are altered.

Sample Z-80 Programming

See $DELAY.

Sample BASIC Programming

```
10 POKE 16526,105: POKE 16527,0          'LSB/MSB
20 X = USR(0)                            'DUMMY ARGUMENT
```

$KBCHAR — 43/X'002B'

Get a Keyboard Character if Available

This subroutine checks the keyboard for a character. The character (if any) is not displayed.

Entry Conditions

None

Exit Conditions

A=ASCII Character. IF A=0, no character was available.

DE is altered.

Sample Z-80 Programming

See $RSINIT.

$KBLINE — 64/X'0040'

Wait for a Line from the Keyboard

This routine gets a full line from the Keyboard. The line is terminated by a carriage return (X'0D') or (BREAK) (X'01'). Characters typed are echoed to the display.

Entry Conditions

B = Maximum length of line. When this many characters are typed, no more will be allowed except for (ENTER) or (BREAK)

(HL) = Storage buffer. Length should be B + 1.

Exit Conditions

C Status = (BREAK) was the terminator.

B = Number of characters entered.

(HL) = Line from keyboard, followed by terminating character.

DE is altered.

Sample Z-80 Programming

See $CSHWR.

$KBWAIT — 73/X'0049'

Wait for a Keyboard Character

This routine scans the keyboard until a key is pressed. If (BREAK) is pressed, it will be returned in A like any other key. The character typed is not echoed to the Display.

Entry Conditions

None

Exit Conditions

A = Keyboard character

DE is altered.

Sample Z-80 Programming

See $CSHWR.

$KBBRK — 653/X'028D'

Check for (BREAK) Key Only

This is a fast key scan for the (BREAK) key only. Use it when you want to minimize keyboard scan time without totally locking out the keyboard.

Entry Conditions

None

Exit Conditions

NZ Status = (BREAK) was pressed

A is altered.

$PRCHAR — 59/X'003B'

Output a Character to the Printer

$PRCHAR waits until the Printer is available or until (BREAK) is pressed. If (BREAK) is pressed, $PRCHAR returns to caller.

Entry Conditions

A = ASCII character

Exit Conditions

DE is altered.

Sample Z-80 Programming

```
                          00148 ; PRINTER DEMO
8356    216583            00149          LD      HL,TXT4          (HL)=SAMPLE TEXT
8359    7E                00150 LOOP5    LD      A,(HL)           GET CHAR. INTO A
835A    23                00151          INC     HL               POINT TO NEXT CHAR
835B    CD3B00            00152          CALL    PRCHAR           PRINT CHAR IN A
835E    FE0D              00153          CP      0DH              WAS IT A CARRIAGE RETURN?
8360    20F7              00154          JR      NZ,LOOP5         IF NO, GET NEXT CHAR.
8362    C3191A            00155          JP      READY            IF YES, QUIT
8365    54                00156 TXT4     DEFM    'THIS SENTENCE WILL BE PRINTED'
8382    0D                00157          DEFB    0DH
402D                      00158          END
00000 ASSEMBLY ERRORS
```

$PRSCN — 473/X'01D9'

Print Entire Screen Contents

This routine copies all 1024 characters from the screen to the printer. If the printer is unavailable, it waits until the printer becomes available. If (BREAK) is pressed, $PRSCN returns to the caller.

Entry Conditions

None

Exit Conditions

All registers are altered.

$READY — 6681/X'1A19'

Jump to Model III BASIC "Ready"

To exit from a machine-language program into BASIC's immediate mode, jump to $READY (don't call it).

Entry Conditions

None

Exit Conditions

None

Sample Z-80 Programming

See $CSHIN.

$RESET — 0/X'0000'

Jump to RESET

Jump to this address to re-initialize the entire system starting at the "Cass?" prompt. If a disk controller is present, the Computer will attempt to load TRSDOS. To prevent this from happening, the operator must hold down (BREAK) before this jump is executed.

Entry Conditions

None

Exit Conditions

None

$ROUTE — 108/X'006C'

Change I/0 Device Routing

Entry Conditions

(X'4222') = Two-byte source device ASCII abbreviation: {KI,DO,RI,RO,PR}

(X'4220') = Two-byte destination device ASCII abbreviation. Same set as above.

Exit Conditions

DE is altered.

Sample Programming.

See Chapter 9 in this section.

$RSINIT — 90/X'005A'

Initialize the RS-232-C Interface

When you start the Computer, the RS-232-C interface is initialized to the following characteristics:

Send/Receive Baud Rate: 300
Word length: 8
Parity: None
Stop-Bits: One
Wait for completion of character I/O

To change any of these, you must call $RSINIT.

Entry Conditions

(16888) = Send/Receive Baud Rate Code:
Most significant four bits = send rate
Least significant four bits = receive rate
See the table of baud rate codes in Chapter 8.

(16890) = Wait/Don't Wait Switch
Zero = "Don't Wait"
Non-Zero = "Wait"

(16889) = RS-232-C Characteristics Switch:

Bits	Meaning	Bits	Meaning
7	Parity: 1 = Even 0 = Odd	3	Transmit On/Off 0 = Disable 1 = Enable
6,5	Word Length: 00 = 5 Bits 01 = 6 Bits 10 = 7 Bits 11 = 8 Bits	2	Data Terminal Ready 0 = No 1 = Yes
		1	Request To Send 0 = No 1 = Yes
5	Stop Bits: 0 = 1 Bit 1 = 2 Bits		
4	Parity On/Off 0 = Parity 1 = No Parity		

Exit Conditions

DE is altered.

Sample Z-80 Program

```
                        00127 ; TERMINAL PROGRAM FOR DEMO OF RS-232-C CALLS, $KBCHAR AND $VDCHAR
                        00128 ;
                        00129 ; ASSUME 16888 & 16889 CONTAIN THE PROPER INITIALIZATION VALUES
                        00130 ;
832C    AF              00131          XOR     A                  ZERO A TO SELECT "DON'T WAIT"
832D    32FA41          00132          LD      (16890),A
8330    CD5A00          00133          CALL    RSINIT
8333    CDC901          00134          CALL VDCLS
8336    CD2B00          00135 KEYIN    CALL    KBCHAR             CHECK KEYBOARD
8339    FE00            00136          CP      0
833B    2806            00137          JR      Z,RSIN             IF NOTHING, CHECK RS232
833D    CD3300          00138          CALL    VDCHAR             SELF-ECHO
8340    CD5500          00139          CALL    RSTX               SEND IT TO RS232
8343    21E841          00140 RSIN     LD      HL,16872           (HL)=CHAR.INPUT BUFFER
8346    CD5000          00141          CALL    RSRCV              CHECK FOR RS232 INPUT
8349    7E              00142          LD      A,(HL)             GET BUFFER CONTENTS
834A    FE00            00143          CP      0
834C    28E8            00144          JR      Z,KEYIN            IF NOTHING, CHECK KB
834E    CD3300          00145          CALL    VDCHAR             ELSE DISPLAY IT
8351    18E3            00146          JR      KEYIN              CHECK KB
8353    C3191A          00147          JP      READY              RETURN TO BASIC
```

$RSRCV — 80/X'0050'

Receive a Character from the RS-232-C Interface

If RS-232-C Wait is enabled, this routine waits for a character to be received, or until (BREAK) is pressed.

If Wait is not enabled, it returns whether or not a character is received.

Entry Conditions

None

Exit Conditions

(16872) = Character received. Zero indicates no character.

DE is altered.

Sample Z-80 Programming

See $RSINIT.

$RSTX — 85/X'0055'

Transmit a Character to the RS-232-C Interface

If RS-232-C Wait is enabled, this routine waits until the character is transmitted or until (BREAK) is pressed.

If Wait is not enabled, it returns whether or not a character is transmitted.

Entry Conditions

A = Character

Exit Conditions

Z Status = No character sent

DE is altered.

Sample Z-80 Programming

See $RSINIT.

$SETCAS — 12354/X'3042'

Prompt User to Set Cassette Baud Rate

This call repeats the first question in the Model III start-up dialog. It displays the prompt:

```
Cass?
```

on the next line of the display, and waits for the operator to type "H" (high—1500 baud) or "L" (low—500) or (ENTER) (default to high).

Upon return from the call, the cassette rate is set accordingly.

Entry Conditions

None

Exit Conditions

All registers are altered.

Sample Z-80 Programming

See $CSHWR.

$TIME — 12342/X'3036'

Get the Time

Entry Conditions

(HL) = Eight-byte output buffer

Exit Conditions

(HL) = Time in this format:
HR:MN:SS

All other registers are altered.

Sample Z-80 Programming

See $DATE.

$VDCHAR — 51/X'0033'

Display a Character

This subroutine displays a character at the current cursor location.

Entry Conditions

A = ASCII character

Exit Conditions

DE is altered.

Sample Z-80 Programming

See $DELAY.

$VDCLS — 457/X′01C9′

Clear the Video Display Screen

Entry Conditions

None

Exit Conditions

All registers are altered.

Sample Z-80 Program

See $CSHWR.

$VDLINE — 539/X′021B′

Display a Line

This subroutine displays a line. The line **must** be terminated with an ASCII ETX (X′03′) or carriage return (X′0D′). If the terminator is a carriage return, it will be printed; if it is an ETX, it will not be printed. This allows VDLINE to position the cursor to the beginning of the next line or leave it at the position after the last text character.

Entry Conditions

(HL) = Output text, terminated by X′03′ or X′0D′.

Exit Conditions

(HL) = First character after the terminator.

DE is altered.

Sample Z-80 Programming

See $CSHWR.

(BREAK) Processing

The (BREAK) key is intercepted during keyboard scan operations. The Computer transfers control to a three-byte jump vector in RAM (hex values: C3 lsb msb). For special applications, you may change the jump vector addresses to allow your own program to handle the (BREAK) key.

The keyscan (BREAK) jump vector is located at 16396 (X'400C').

Register contents on entry to the jump vector

DE = Modified by the Computer

(SP) = The return address of the interrupted program. That is, a RET will transfer control to the point at which the program was interrupted.

Sample BASIC Programming

Run this BASIC program to disable (BREAK).

```
10 POKE 16396,175          '175 = Z-80 "XOR A" CODE
20 POKE 16397,201          '201 = Z-80 "RET" CODE
```

Run this BASIC program to enable the (BREAK) key.

```
10 POKE 16396,201          'Z-80 "RET" CODE
```

Memory Map

Decimal Address	Contents	Hexadecimal Address
0	12 K ROM Model III BASIC	0
12288	2 K ROM for System Use	3000
14336	Keyboard Matrix	3800
15360	Memory-Mapped Video Display: Upper left corner = 15360 + 0. Lower right corner = 15360 + 1023.	3C00
16384	Reserved for System Use	4000
17129	User Memory For Program and Data	42E9
32767 49151 65535	"16K RAM" ends here. "32K RAM" ends here. "48K RAM" ends here.	7FFF BFFF FFFF

Summary of Important ROM Addresses

Address			
Dec	Hex	Contents	Function
0	0000	$RESET	System reset
43	002B	$KBCHAR	Check for keyboard character
51	0033	$VDCHAR	Display a character
59	003B	$PRCHAR	Print a character
64	0040	$KBLINE	Wait for a keyboard line
73	0049	$KBWAIT	Wait for a keyboard character
80	0050	$RSRCV	Receive character from RS-232-C
85	0055	$RSTX	Transmit character to RS-232-C
90	005A	$RSINIT	Initialize RS-232-C
96	0060	$DELAY	Delay for a specified time
105	0069	$INITIO	Initialize all I/O drivers
108	006C	$ROUTE	Route I/O
457	01C9	$VDCLS	Clear the screen
473	01D9	$PRSCN	Print screen contents
504	01F8	$CSOFF	Turn off cassette
539	021B	$VDLINE	Display a line
565	0235	$CSIN	Input a cassette byte
612	0264	$CSOUT	Output a cassette byte
647	0287	$CSHWR	Write the cassette header
653	028D	$KBBRK	Check for (BREAK) key only
662	0296	$CSHIN	Read the cassette header
664	0298	$CLKON	Turn on the clock display
673	02A1	$CLKOFF	Turn off the clock display
6681	1A19	$READY	Jump to BASIC "Ready"
12339	3033	$DATE	Get the date
12342	3036	$TIME	Get the time
12354	3042	$SETCAS	Set cassette baud rate
14312	37E8	$PRSTAT	Printer status (Read Only) "Go" only if: Bit 7 = 0 "NOT BUSY" Bit 6 = 0 "NOT OUT OF PAPER" Bit 5 = 1 "DEVICE SELECT" Bit 4 = 1 "NOT PRINTER FAULT" Bits 3,2,1 and 0 are not used.

Summary of Important RAM Addresses

Address Dec	Address Hex	Contents	Initial Contents
16396	400C	(BREAK) Jump Vector Keyboard scan operations Three bytes	C9 xx xx
16409	4019	Caps Lock Switch 0 = "Upper and Lower Case" Not 0 = "Caps Only"	"Caps"
16412	401C	Cursor Blink Switch 0 = "Blink" Non-Zero = "No-Blink"	"Blink"
16416	4020	Cursor Address Two bytes: LSB, MSB	N/A
16419	4023	Cursor Character ASCII Code 32 — 255	176
16424	4028	Maximum Lines/Page plus one	67
16425	4029	Number of lines printed plus one	1
16427	402B	Line Printer Max. Line length less two. 255 = "No Maximum"	"No Max"
16872	41E8	$RSRCV Input Buffer One byte	0
16880	41F0	$RSTX Output Buffer One byte	0
16888	41F8	$RSINIT Baud Rate Code TX Code = Most Sig. Nibble RCV Code = Least Sig. Nibble	85
16889	41F9	$RSINIT Parity/Word Length/ Stop-Bit Code	108
16890	41FA	$RSINIT WAIT Switch 0 = "Don't Wait" Non-Zero = "Wait"	"Wait"
16913	4211	Cassette Baud Rate Switch 0 = 500 Baud Non-Zero = 1500 Baud	N/A

Address Dec	Address Hex	Contents	Initial Contents
16916	4214	Video Display Scroll Protect From 0 to 7. Greater values are interpreted in modulo 8	0
16919	4217	Time-Date Six binary bytes: SS MM HH YY DD MM	00:00:00 00/00/00
16928	4220	$ROUTE Destination Device Two-byte I/O designator	N/A
16930	4222	$ROUTE Source Device Two-byte I/O designator	N/A

13 / Troubleshooting And Maintenance

If you have problems operating your TRS-80, please check the following table of symptoms and cures. It's also possible that you have not followed the instructions correctly.

If you can't solve the problem, take the unit in to your local Radio Shack. We'll have it fixed and returned to you ASAP!

Symptom	Possible Cause. Cure.
The Cass? message does not appear when you turn on the Computer.	1. No AC power. Check power cord connection to Computer and all peripherals. 2. Incorrect power-up sequence. 3. Peripheral device (e.g., printer) is not connected properly. Recheck connection. 4. Disk system. To operate without a TRSDOS diskette, hold down (BREAK) while you reset or power on. 5. Video Display needs adjustment. Check Brightness and Contrast controls.

Symptom	Possible Cause. Cure.
Can't get a cassette program to load.	1. Improper cassette connection. Check connection instructions in cassette owner's manual. 2. Cassette load speed does not match the speed of the recorded tape. Model I Level II BASIC programs are always Low (500 baud). Model III programs may be either High (1500) or Low. 3. Incorrect volume setting. Try another volume setting. 4. Information on tape may have been garbled due to static electricity discharge, magnetic field, or tape deterioration. Try to load duplicate copy, if available.
Computer "hangs up" during normal operation, requiring reset or power-off/on	1. Fluctuations in the AC power supply. See AC Power Sources, below. 2. Defective or improperly installed connector. Check all connection cables to see that they are securely attached and that they are not frayed or broken. 3. Programming. Re-check the program.

AC Power Sources

Computers are sensitive to fluctuations in the power supply at the wall socket. This is rarely a problem unless you are operating in the vicinity of heavy electrical machinery. The power source may also be unstable if some appliance or office machine in the vicinity has a defective switch which arcs when turned on or off.

Your Model III TRS-80 is equipped with a specially designed, built-in AC line filter. It should eliminate the effects of ordinary power-line fluctuations.

However, if the fluctuations are severe, you may need to take some or all of the following steps:

- Install bypass or isolation devices in the problem-causing devices
- Fix or replace any defective (arcing) switches
- Install a separate power-line for the Computer
- Install a special line filter designed for computers and other sensitive electronic equipment

Power line problems are rare and many times can be prevented by proper choice of installation location. The more complex the system and the more serious the application, the more consideration you should give to providing an ideal power source for your Computer.

Maintenance

Your Computer requires little maintenance. It's a good idea to keep it clean and free of dust build-up. This is especially important for the keyboard. Radio Shack sells a custom-designed Model III dust cover you may find helpful.

If you need to clean the Computer case, use a damp, lint-free cloth.

The peripheral devices (cassette recorder, line printer, etc.) may require more maintenance. Check the owner's manual for each peripheral in your system.

14 / Specifications

AC Power Supply

This applies to non-disk systems only. For disk systems, see the Disk System Owner's Manual.

Power Requirements	105 - 130 VAC, 60 Hz (240 VAC, 50 Hz Australian) (220 VAC, 50 Hz European)
Current Drain	0.83 Amps RMS

Microprocessor

Type	Z-80
Clock Rate	2.02752 MHz

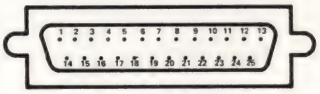
RS-232-C Interface

Standard RS-232-C Signal		Pin #
PG	Protective Ground	1
TD	Transmit Data	2
RD	Receive Data	3
RTS	Request To Send	4
CTS	Clear To Send	5
DSR	Data Set Ready	6
SG	Signal Ground	7
CD	Carrier Detect	8
DTR	Data Terminal Ready	20
RI	Ring Indicator	22
STD*	Secondary Transmit Data	14
SUN*	Secondary Unassigned	18
SRTS*	Secondary Request To Send	19

***Note:** These signals are not used for the secondary functions, but are reserved for future use.

RS-232-C Pin Location

Looking from the outside at the RS-232-C jack on the Model III Computer:



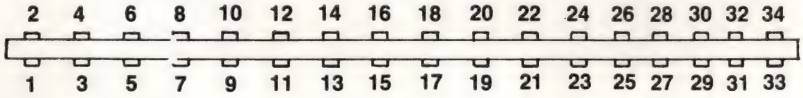
Parallel Printer Interface

Signal	Function	Pin #
STROBE*	1.5 μS pulse to clock the data from processor to printer	1
DATA 0	Bit 0 (lsb) of output data byte	3
DATA 1	Bit 1 of output data byte	5
DATA 2	Bit 2 of output data byte	7
DATA 3	Bit 3 of output data byte	9
DATA 4	Bit 4 of output data byte	11
DATA 5	Bit 5 of output data byte	13
DATA 6	Bit 6 of output data byte	15
DATA 7	Bit 7 (msb) of output data byte	17
BUSY	Input to Computer from Printer, high indicates busy	21
PAPER EMPTY	Input to Computer from Printer, high indicates no paper — if Printer doesn't provide this, signal is forced low	23
SELECT	Input to Computer from Printer, high indicates device selected	25
FAULT*	Input to Computer from Printer, low indicates fault (paper empty, light detect, deselect, etc.)	28
GROUND	Common signal ground	2,4,6,8,10 12,14,16,18, 20,22,24,27, 31,33,34
NC	Not connected or not used	26,29,30,32

*These signals are active-low.

Printer Pin Location

Looking from the bottom rear at the printer card-edge connector as in Figure 1 on 2/2:

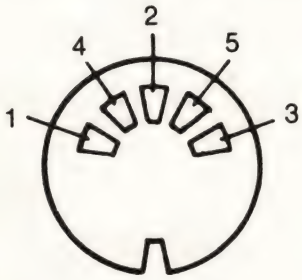


Cassette Interface

Suggested Input Level for Playback from Recorder	1 to 5 Volts peak-to-peak at a minimum impedance of 220 Ohms
Typical Computer Output Level to Recorder	800 mV peak-to-peak at 1 K Ohm
Remote On/Off Switching Capability	0.5 A maximum at 6 VDC

Cassette Jack Pin Location

Looking at the outside of the cassette jack on the Computer:



1. **Remote Control**
2. **Signal Ground**
3. **Remote Control**
4. **Input from Recorder's Earphone Jack**
5. **Output to Recorder's Aux or Mic Jack**

Section 2:
BASIC Language

```
960 FOR K = 1 TO N3
970 FOR J = 1 TO N2
980 FOR I = 1 TO N1
990 C(I,J,K) = A(I,J,K)
1000 NEXT I
1010 NEXT J
1020 NEXT K
1030 END
```

1 / BASIC Concepts

This chapter gives an in-depth description of how to use the full power of Model III BASIC. Programmers require this information in order to build powerful and efficient programs. However, if you are still somewhat of a novice, you might want to skip this chapter for now, keeping in mind that the information is here when you need it.

This chapter is divided into four sections:

1. Overview — Elements of a Program. This section defines many of the terms we will be using in the chapter.

2. How BASIC Handles Data. Here we discuss how BASIC classifies and stores data. This will show you how to get BASIC to store your data in its most efficient format.

3. How BASIC Manipulates Data. This will give you an overview of all the different operators and functions you can use to manipulate and test your data.

4. How to Construct an Expression. Understanding this topic will help you form powerful statements instead of using many short ones.

Overview — Elements of a Program

This overview defines the elements of a program:

The **program** itself, which consists of . . .

Statements, which may consist of . . .

Expressions

We will refer to these terms during the rest of this chapter.

Program

A program is made up of one or more numbered lines. Each line contains one or more BASIC statements. BASIC allows line numbers from 0 to 65529 inclusive. You may include up to 255* characters per line, including the line number. You may also have two or more statements to a line, separated by colons.

*You can only type in 240 characters for new lines; using the Edit Mode, you can add the extra 15 characters.

Here is a sample program:

Line number — BASIC statement — Colon between statements — BASIC statement

```
100 CLS: PRINT "NORMAL MODE..."
110 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
120 FOR I = 1 TO 1000: NEXT I
130 CLS: PRINT CHR$(23); "DOUBLE-SIZE MODE..."
140 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
150 END
```

When BASIC executes a program, it handles the statements one at a time, starting at the first and proceeding to the last. Some statements, such as GOTO, ON . . . GOTO, GOSUB, change this sequence.

Statements

A statement is a complete instruction to BASIC, telling the Computer to perform specific operations. For example:

```
GOTO 100
```

Tells the Computer to perform the operations of (1) locating line 100 and (2) executing the statement on that line.

```
END
```

Tells the Computer to perform the operation of ending execution of the program.

Many statements instruct the computer to perform operations with data. For example, in the statement:

```
PRINT "SEPTEMBER  REPORT"
```

the data is SEPTEMBER REPORT. The statement instructs the Computer to print the data inside the quotes.

Expressions

An expression is actually a general term for data. There are four types of expressions:

1. **Numeric expressions,** which are composed of numeric data. Examples:

```
(1 + 5.2) / 3
D
5 * B
3.7682
ABS(X) + RND(0)
SIN(3 + E)
```

2. **String expressions,** which are composed of character data. Examples:

```
A$
"STRING"
"STRING" + "DATA"
MO$ + "DATA"
MID$(A$,2,5) + MID$("MAN",1,2)
M$ + A$ + B$
```

3. Relational expressions, which test the relationship between two expressions. Examples:

```
A = 1
A$ > B$
```

4. Logical expressions, which test the logical relationship between two expressions. Examples:

```
A$ = "YES"  AND  B$ = "NO"
C > 5  OR  M < B  OR  O > 2
578  AND  452
```

Functions

Functions are automatic subroutines. Most BASIC functions perform computations on data. Some serve a special purpose such as controlling the video display or providing data on the status of the computer. You may use functions in the same manner that you use any data — as part of a statement.

These are some of BASIC's functions:

```
INT
ABS
STRING$
```

How Basic Handles Data

Model III BASIC offers several different methods of handling your data. Using these methods properly can greatly improve the efficiency of your program. In this section we will discuss:

1. Ways of Representing Data
   a. Constants
   b. Variables
2. How BASIC Stores Data
   a. Numeric (integer, single precision, double precision)
   b. String
3. How BASIC Classifies Constants
4. How BASIC Classifies Variables
5. How BASIC Converts Data

Ways of Representing Data

BASIC recognizes data in two forms — either directly, as constants, or by reference to a memory location, as variables.

Constants

All data is input into a program as ''constants'' — values which are not subject to change. For example, the statement:

```
PRINT  "1 PLUS 1 EQUALS";  2
```

contains one string constant,

```
1 PLUS 1 EQUALS
```

and one numeric constant

```
2
```

In these examples, the constants ''input'' to the PRINT statement. They tell PRINT what data to print on the Display.

These are more examples of constants:

3.14159	"L. O. SMITH"
1.775E+3	"0123456789ABCDEF"
"NAME TITLE"	–123.45E-8
57	"AGE"

Variables

A variable is a place in memory — a sort of box or pigeonhole — where data is stored. Unlike a constant, a variable's value can change. This allows you to write programs dealing with changing quantities. For example, in the statement:

```
A$ = "OCCUPATION"
```

The variable A$ now contains the data OCCUPATION. However, if this statement appeared later in the program:

```
A$ = "FINANCE"
```

The variable A$ would no longer contain OCCUPATION. It would now contain the data FINANCE.

Variable Names

In BASIC, variables are represented by names. Variable names must begin with a letter, A through Z. This letter may be followed by one more character — either a digit or a letter.

For example

AM A A1 B1 AB

are all valid and distinct variable names.

Variable names may be longer than two characters. However, only the first two characters are significant in BASIC.

For example:

SUM SU SUPERNUMERARY

are all treated as the same variable by BASIC.

Reserved Words

Certain combinations of letters are reserved as BASIC keywords, and cannot be used in variable names. For example:

OR LAND LENGTH MIFFED

cannot be used as variable names, because they contain the reserved of OR, AND, LEN, and IF, respectively.

See the Appendix for a list of reserved words.

Simple and Subscripted Variables

All of the variables mentioned above are simple variables. They can only refer to one data item.

Variables may also be subscripted so that an entire list of data can be stored under one variable name. This method of data storage is called an **array**. For example, an array named A may contain these elements (subscripted variables):

A(0) A(1) A(2) A(3) A(4)

You may use each of these elements to store a separate data item, such as:

```
A(0) = 5.3
A(1) = 7.2
A(2) = 8.3
A(3) = 6.8
A(4) = 3.7
```

In this example, array A is a one-dimensional array, since each element contains only one subscript. An array may also be two-dimensional, with each element containing two subscripts. For example, a two-dimensional array named X could contain these elements:

```
X(0,0) = 8.6        X(0,1) = 3.5
X(1,0) = 7.3        X(1,1) = 32.6
```

With BASIC, you may have as many dimensions in your array as you would like. Here is an example of a three-dimensional array named L which contains these 8 elements:

```
L(0,0,0) = 35233    L(0,1,0) = 96522
L(0,0,1) = 52000    L(0,1,1) = 10255

L(1,0,0) = 33333    L(1,1,0) = 96253
L(1,0,1) = 53853    L(1,1,1) = 79654
```

BASIC assumes that all arrays contain 11 elements in each dimension. If you want more elements you must use the DIM statement at the beginning of your program to dimension the array.

For example, to dimension array L, put this line at the beginning of the program:

```
DIM L (1, 1, 1)
```

to allow room for two elements in the first dimension; two in the second; and two in the third for a total of $2 * 2 * 2 = 8$ elements.

See the Arrays chapter later on in this manual.

How BASIC Stores Data

The way that BASIC stores data determines the amount of memory it will consume and the speed in which BASIC can process it.

Numeric Data

You may get BASIC to store all numbers in your program as either integer, single precision, or double precision. In deciding how to get BASIC to store your numeric data, remember the tradeoffs. Integers are the most efficient and the least precise. Double precision is the most precise and least efficient.

Integers
(Speed and Efficiency, Limited Range)

To be stored as an integer, a number must be whole and in the range of −32768 to 32767. An integer value requires only two bytes of memory for storage. Arithmetic operations are faster when both operands are integers.

For example:

1	32000	−2	500	−12345

can all be stored as integers.

Single-Precision Type
(General Purpose, Full Numeric Range)

Single-precision numbers can include up to 7 significant digits, and can represent normalized values* with exponents up to ±38, i.e., numbers in the range:

$[-1 \times 10^{38}, -1 \times 10^{-38}]\,[1 \times 10^{-38}, 1 \times 10^{38}]$

A single-precision value requires 4 bytes of memory for storage. BASIC assumes a number is single-precision if you do not specify the level of precision.

*In this reference manual, normalized value is one in which exactly one digit appears to the left of the decimal point. For example, 12.3 expressed in normalized form is 1.23 x 10.

For example:

```
10.001      -200034      1.774E6      6.024E-23      123.4567
```

can all be stored as single-precision values.

Note: When used in a decimal number, the symbol E stands for ''single-precision times 10 to the power of...'' Therefore 6.024E-23 represents the single-precision value:

6.024×10^{-23}

Double-Precision Type
(Maximum Precision, Slowest in Computations)

Double-precision numbers can include up to 17 significant digits, and can represent values in the same range as that for single-precision numbers. A double-precision value requires 8 bytes of memory for storage. Arithmetic operations involving at least one double-precision number are slower than the same operations when all operands are single-precision or integer.

For example:

```
1010234578
-8.7777651010
3.1415926535897932
8.00100708D12
```

can all be stored as double-precision values.

Note: When used in a decimal number, the symbol D stands for ''double-precision times 10 to the power of...'' Therefore 8.00100708 D12 represents the value
$8.00100708 \times 10^{12}$

String Data

Strings (sequences of characters) are useful for storing non-numeric information such as names, addresses, text, etc. You may store any ASCII characters as a string. (A list of ASCII characters is in the Appendix).

For example, the data constant:

```
Jack  Brown,  Age  38
```

can be stored as a string of 18 characters. Each character (and blank) in the string is stored as an ASCII code, requiring one byte of storage. BASIC would store the above string constant internally as:

Hex Code	4A	61	63	6B	20	42	72	6F	77	6E	2C	20	41	67	65	20	33	38
ASCII Character	J	a	c	k		B	r	o	w	n	,		A	g	e		3	8

A string can be up to 255 characters long. Strings with length zero are called "null" or "empty".

How BASIC Classifies Constants

When BASIC encounters a data constant in a statement, it must determine the type of the constant: string, integer, single precision, or double precision. First, we will list the rules BASIC uses to classify the constant. Then we will show you how you can override these rules, if you want a constant stored differently:

Rule 1

If the value is enclosed in double-quotes, it is a string. For example:

```
"YES"
"3331 Waverly Way"
"1234567890"
```

the values in quotes are automatically classified as strings.

Rule 2

If the value is not in quotes, it is a number. (An exception to this rule is during data input by an operator, and in DATA lists. See INPUT, INKEY$, and DATA)

For example:

```
123001
1
-7.3214E+6
```

are all numeric data.

Rule 3

Whole numbers in the range of −32768 to 32767 are integers. For example:

```
12350
-12
10012
```

are integer constants.

Rule 4

If the number is not an integer and contains seven or fewer digits, it is single-precision. For example:

```
1234567
-1.23
1.3321
```

are all single-precision.

Rule 5

If the number contains more than seven digits, it is double precision. For example, these numbers:

```
1234567890123456
-1000000000000.1
2.777000321
```

are all double precision.

Type Declaration Tags

You can override BASIC's normal typing criteria by adding the following "tags" to the end of the numeric constant:

! Makes the number single-precision. For example, in the statement:

```
A = 12.345678901234!
```

the constant is classified as single-precision, and shortened to seven digits: 12.34567

E Single-precision exponential format. The E indicates the constant is to be multipled by a specified power of 10. For example:

```
A = 1.2E5
```

stores the single-precision number 120000 in A.

Makes the number double-precision. For example, in statement:

```
PRINT 3#/7
```

the first constant is classified as double-precision before the division takes place.

D Double-precision exponential format. The D indicates the constant is to be multipled by a specified power of 10. For example:

```
A = 1.23456789D - 1
```

The double-precision constant has the value 0.123456789.

How BASIC Classifies Variables

When BASIC encounters a variable name in the program, it classifies it as either a string, integer, single- or double-precision number.

BASIC classifies all variable names as single-precision initially. For example:

```
AB        AMOUNT        XY        L
```

are all single-precision initially. If this is the first line of your program:

```
LP = 1.2
```

BASIC will classify LP as a single-precision variable.

However, you may assign different attributes to variables by using definition statements at the beginning of your program:

DEFINT — Defines variables as integer
DEFDBL — Defines variables as double-precision
DEFSTR — Defines variables as string
DEFSNG — Defines variables as single-precision. (Since BASIC classifies all variables as single-precision initially anyway, you would only need to use DEFSNG if one of the other DEF statements were used.

For example:

```
DEFSTR  L
```

makes BASIC classify all variables which start with L as string variables. After this statement, the variables:

```
L        LP        LAST
```

can all hold string values only.

Type Declaration Tags

As with constants, you can always override the type of a variable name by adding a type declaration tag at the end. There are four type declaration tags for variables:

%	Integer
!	Single-precision
#	Double-precision n
$	String

For example:

```
I%        FT%        NUM%        COUNTER%
```

are all integer variables, **regardless** of what attributes have been assigned to the letters I, F, N and C.

```
T!        RY!        QUAN!        PERCENT!
```

are all single-precision variables, **regardless** of what attributes have been assigned to the letters T, R, Q and P.

```
X#        RR#        PREV#        LSTNUM#
```

are all double-precision variables, **regardless** of what attributes have been assigned to the letters X, R, P and L.

```
Q$        CA$        WRD$        ENTRY$
```

are all string variables, **regardless** of what attributes have been assigned to the letters Q, C, W and E.

Note that any given variable name can represent four different variables. For example:

```
A5#        A5!        A5%        A5$
```

are all valid and **distinct** variable names.

One further implication of type declaration: Any variable name used without a tag is equivalent to the same variable name used with one of the four tags. For example, after the statement:

```
DEFSTR    C
```

the variable referenced by the name C1 is identical to the variable referenced by the name C1$.

How BASIC Converts Numeric Data

Often your program might ask BASIC to assign one type of constant to a different type of variable. For example:

```
A% = 2.34
```

In this example, BASIC must first convert the single precision constant 2.34 to an integer in order to assign it to the integer variable A%.

You might also want to convert one type of variable to a different type, such as:

```
A# = A%
A! = A#
A! = A%
```

The conversion procedures are listed on the following pages.

Single- or double-precision to integer type

BASIC returns the largest integer that is not greater than the original value.

Note: The original value must be greater than or equal to -32768, and less than 32768.

Examples

```
A% = -10.5
```

Assigns A% the value -11.

```
A% = 32767.9
```

Assigns A% the value 32767.

```
A% = 2.5D3
```

Assigns A% the value 2500.

```
A% = -123.45678901234578
```

Assigns A% the value -124.

```
A% = -32768.1
```

Produces an Overflow Error (out of integer range).

Integer to single- or double-precision

No error is introduced. The converted value looks like the original value with zeros to the right of the decimal place.

Examples

```
A# = 32767
```

Stores 32767.000000000000 in A#.

```
A! = -1234
```

Stores -1234.000 in A!.

Double- to single-precision

This involves converting a number with up to 17 significant digits into a number with no more than seven. BASIC chops off (truncates) the least significant digits to produce a seven-digit number. Before Printing such a number, BASIC rounds it off (4/5 rounding) to six digits.

Examples

```
A! = 1.234567890124567
```

Stores 1.234567 in A! However, the statement:

```
PRINT A!
```

will display the value 1.23457, because only six digits are displayed. The full seven digits are stored in memory.

```
A! = 1.3333333333333333
```

Stores 1.333333 in A!.

Single- to double-precision

To make this conversion, BASIC simply adds trailing zeros to the single-precision number. If the original value has an exact binary representation in single-precision format, no error will be introduced. For example:

```
A# = 1.5
```

Stores 1.5000000000000 in A#, since 1.5 *does* have an exact binary representation.

However, for numbers which have no exact binary representation, an error is introduced when zeros are added. For example:

```
A# = 1.3
```

Stores 1.299999952316284 in A#.

Because most fractional numbers do not have an exact binary representation, you should keep such conversions out of your programs. For example, whenever you assign a constant value to a double-precision variable, you can force the constant to be double-precision:

```
A# = 1.3#        A# = 1.3D
```

Both store 1.3 in A#.

Here is a special technique for converting single-precision to double-precision, without introducing an error into the double-precision value. It is useful when the single-precision value is stored in a variable.

Take the single-precision variable, convert it to a string with STR$, then convert the resultant string back into a number with VAL. That is, use:

VAL (STR$ (*single-precision variable*))

For example, the following program:

```
10  A! = 1.3
20  A# = A!
30  PRINT A#
```

prints a value of:

```
1.299999952316284
```

Compare with this program:

```
10 A! = 1.3
20 A# = VAL (STR$(A!))
30 PRINT A#
```

which prints a value of:

```
1.3
```

The conversion in line 20 causes the value in A! to be stored accurately in double-precision variable A#.

Illegal Conversions

BASIC cannot automatically convert numeric values to string, or vice versa. For example, the statements:

```
A$ = 1234
A% = "1234"
```

are illegal. (Use STR$ and VAL to accomplish such conversions.)

How BASIC Manipulates Data

You have many fast methods you may use to get BASIC to count, sort, test and rearrange your data. These methods fall into two categories:

1. Operators
   a. numeric
   b. string
   c. relational
   d. logical

2. Functions

Operators

An operator is the single symbol or word which signifies some action to be taken on either one or two specified values referred to as operands.

In general, an operator is used like this:

operand-1 operator operand-2

operand-1 and *-2* can be expressions. A few operations take only one operand, and are used like this:

operator operand

Examples:

```
6 + 2
```

The addition operator + connects or relates its two operands, 6 and 2, to produce the result 8.

```
-5
```

The negation operator − acts on a single operand 5 to produce the result negative 5.

Neither 6 + 2 or −5 can stand alone; they must be used in statements to be meaningful to BASIC. For example:

```
A = 6 + 2
PRINT -5
```

Operators fall into four categories:

- Numeric
- String
- Relational
- Logical

based on the kinds of operands they require and the results they produce.

Numeric Operators

Numeric Operators are used in numeric expressions. Their operands must always be numeric, and the result they produce is one numeric data item.

In the descriptions below, we use the terms integer, single-precision, and double-precision operations. Integer operations involve two-byte operands, single-precision operations involve four-byte operands, and double-precision operations involve eight-byte operands. The more bytes involved, the slower the operation.

There are five different numeric operators. Two of them, sign + and sign −, are unary, that is, they have only one operand. A sign operator has no effect on the precision of its operand.

For example, in the statement:

```
PRINT -77, +77
```

the sign operators − and + produce the values negative 77 and positive 77, respectively.

Note: When no sign operator appears in front of a numeric term, + is assumed.

The other numeric operators are all binary, that is, they all take two operands. These operators are:

+	Addition
−	Subtraction
*	Multiplication
/	Division
[or ↑	Exponentiation. Press the ↑ key to type in this operator.

Addition

The + operator is the symbol for addition. The addition is done with the precision of the more precise operand (the less precise operand is converted).

For example, when one operand is integer type and the other is single-precision, the integer is converted to single-precision and four-byte addition is done. When one operand is single-precision and the other is double-precision, the single-precision number is converted to double-precision and eight-byte addition is done.

Examples:

```
PRINT  2+3
```

Integer addition.

```
PRINT  3.1 + 3
```

Single-precision addition.

```
PRINT 1.2345678901234567 + 1
```

Double-precision addition.

Subtraction

The − operator is the symbol for subtraction. As with addition, the operation is done with the precision of the more precise operand (the less precise operand is converted).

Examples:

```
PRINT  33 − 11
```

Integer subtraction.

```
PRINT  33 − 11.1
```

Single-precision subtraction.

```
PRINT  12.345678901234567 − 11
```

Double-precision subtraction.

Multiplication

The * operator is the symbol for multiplication. Once again, the operation is done with the precision of the more precise operand (the less precise operand is converted).
Examples:

```
PRINT  33 * 11
```

Integer multiplication.

```
PRINT  33 * 11.1
```

Single-precision multiplication.

```
PRINT  12.345678901234567 * 11
```

Double-precision multiplication.

Division

The / symbol is used to indicate ordinary division. Both operands are converted to single or double-precision, depending on their original precision:

- If either operand is double-precision, then both are converted to double-precision and eight-byte division is performed.
- If neither operand is double-precision, then both are converted to single-precision and four-byte division is performed.

Examples:

```
PRINT  3/4
```

Single-precision division.

```
PRINT  3.8 / 4
```

Single-Precision division.

```
PRINT  3 / 1.2345678901234567
```

Double-precision division.

Exponentiation

The symbol [denotes exponentiation. It converts both its operands to single-precision, and returns a single-precision result.

Note: To enter the [operator, press ⇧.

For example:

```
PRINT 6[.3
```

prints 6 to the .3 power.

String Operator

BASIC has a string operator (+) which allows you to concatenate (link) two strings into one. This operator should be used as part of a string expression. The operands are both strings and the resulting value is one piece of string data.

The + operator links the string on the right of the sign to the string on the left. For example:

```
PRINT  "CATS" + "LOVE" + "MICE"
```

prints:

```
CATSLOVEMICE
```

Since BASIC does not allow one string to be longer than 255 characters, you will get an error if your resulting string is too long.

Relational Operators

Relational operators compare two numerical or two string expressions to form a relational expression. This expression reports whether the comparison you set up in your program is true or false. It will return a − 1 if the relation is true; a 0 if it is false.

Numeric Relations

This is the meaning of the operators when you use them to compare numeric expressions:

<	Less than
>	Greater than
=	Equal to
<> or ><	Not equal to
=< or <=	Less than or equal to
=> or >=	Greater than or equal to

Examples of true relational expressions:

```
1<2
2<>5
2<=5
2<=2
5>2
7=7
```

String Relations

The relational operators for string expressions are the same as above, although their meanings are slightly different. Instead of comparing numerical magnitudes, the operators compare their ASCII sequence. This allows you to sort string data:

<	Precedes
>	Follows
=	Has the same precedence
>< or <>	Does not have the same precedence
<=	Precedes or has the same precedence
>=	Follows or has the same precedence

BASIC compares the string expressions on a character-by-character basis. When it finds a non-matching character, it checks to see which character has the lower ASCII code. The character with the lower ASCII code is the smaller (precedent) of the two strings.

Note: The appendix contains a listing of ASCII codes for each character.

Examples of true relational expressions:

```
"A" < "B"
```

The ASCII code for A is decimal 65; for B it's 66.

```
"CODE" < "COOL"
```

The ASCII code for O is 79; for D it's 68.

If while making the comparison, BASIC reaches the end of one string before finding non-matching characters, the shorter string is the precedent. For example:

```
"TRAIL"  <  "TRAILER"
```

Leading and trailing blanks are significant. For example:

```
" A"  <  "A"
```

ASCII for the space character is 32; for A, it's 65.

```
"Z-80" < "Z-80A"
```

The string on the left is four characters long; the string on the right is five.

How to Use Relational Expressions

Normally, relational expressions are used as the test in an IF/THEN statement. For example:

```
IF  A = 1  THEN  PRINT  "CORRECT"
```

BASIC tests to see if A is equal to 1. If it is, BASIC prints the message.

```
IF  A$  <  B$  THEN  50
```

If string A$ alphabetically precedes string B$, then the program branches to line 50.

```
IF  R$ = "YES"  THEN  PRINT  A$
```

If R$ equals YES then the message stored as A$ is printed.

However, you may also use relational expressions simply to return the true or false results of a test. For example:

```
PRINT  7 = 7
```

Prints − 1 since the relation tested is true.

```
PRINT  "A"  >  "B"
```

Prints 0 because the relation tested is false.

Logical Operators

Logical operators make logical comparisons. Normally, they are used in IF/THEN statements to make a logical test between two or more relations. For example:

```
IF A = 1      OR  C = 2      THEN  PRINT  X
```

The logical operator, OR, compares the two relations A = 1 and C = 2.

Logical operators may also be used to make bit-comparisons of two numeric expressions.

For this application, BASIC does a bit-by-bit comparison of the two operands, according to predefined rules for the specific operator.

Note: The operands are converted to integer type, stored internally as 16-bit, two's complement numbers. To understand the results of bit-by-bit comparisons, you need to keep this in mind.

The following table summarizes the action of Boolean operators in bit manipulation.

Operator	Meaning of Operation	First Operand	Second Operand	Result
AND	When both bits are 1, the result will be 1. Otherwise, the result will be 0.	1 1 0 0	1 0 1 0	1 0 0 0
OR	Result will be 1 unless both bits are 0.	1 1 0 0	1 0 1 0	1 1 1 0
NOT	Result is opposite of bit.	1 0		0 1

Hierarchy of Operators

When your expressions have multiple operators, BASIC performs the operations according to a well-defined hierarchy, so that results are always predictable.

Parentheses

When a complex expression includes parentheses, BASIC always evaluates the expressions inside the parentheses before evaluating the rest of the expression. For example, the expression:

```
8 - (3-2)
```

is evaluated like this:

```
3 - 2 = 1
  8 - 1 = 7
```

With nested parentheses, BASIC starts evaluating the innermost level first and works outward. For example:

```
4*(2 - (3 - 4))
```

is evaluated like this:

```
3 - 4 = -1
2 - (-1) = 3
  4*3 = 12
```

Order of Operations

When evaluating a sequence of operations on the same level of parenthesis, BASIC uses a hierarchy to determine what operation to do first.

The two listings below show the hierarchy BASIC uses. Operators are shown in decreasing order of precedence. Operators listed in the same entry in the table have the same precedence and are executed as encountered **from left to right:**

Numerical operations:

[or (Exponentiation)
+, − (Unary sign operands [**not** addition and subtraction])
*, /
+, − (Addition and subtraction)
<, >, =, <=, >=, <>
NOT
AND
OR

String operations:

+
<, >, =, <=, >=, <>

For example, in the line:

```
X*X + 5[2.8
```

BASIC will find the value of 5 to the 2.8 power. Next, it will multiply X * X, and finally add this value to the value of 5 to the 2.8. If you want BASIC to perform the indicated operations in a different order, you must add parentheses. For example:

```
X*(X + 5[2.8)
```

or

```
X*(X + 5)[2.8
```

Here's another example:

```
IF  X = 0   OR   Y > 0   AND   Z = 1   THEN  255
```

The relational operators = and > have the highest precedence, so BASIC performs them first, one after the next, from left to right. Then the logical operations are performed. AND has a higher precedence than OR, so BASIC performs the AND operation before OR.

If the above line looks confusing because you can't remember which operator is precedent over which, then you can use parentheses to make the sequence obvious:

```
IF  X = 0   OR   ((Y > 0)   AND   (Z = 1))   THEN  255
```

Functions

A function is a built-in sequence of operations which BASIC will perform on data. A function is actually a subroutine which usually returns a data item. BASIC functions save you from having to write a BASIC routine, and they operate faster than a BASIC routine would.

A function consists of a keyword which is usually followed by the data that you specify. This data is always enclosed in parentheses; if more than one data item is required, the items are separated by commas.

If the data required is termed "number" you may insert any numerical expression. If it is termed "string" you may insert a string expression.

Examples:

```
SQR (A + 6)
```

Tells BASIC to compute the square root of (A + 6).

```
MID$ (A$, 3, 2)
```

Tells BASIC to return a substring of the string A$, starting with the third character, with a length of 2.

Functions cannot stand alone in a BASIC program. Instead they are used in the same way you use expressions — as the data in a statement.

For example

```
A = SQR (7)
```

Assigns A the data returned as the square root of 7.

```
PRINT  MID$ (A$, 3, 2)
```

Prints the substring of A$ starting at the third character and two characters long.

If the function returns numeric data, it is a numeric function and may be used in a numeric expression. If it returns string data, it is a string function and may be used in a string expression.

How to Construct an Expression

Understanding how to construct an expression will help you put together powerful statements – instead of using many short ones. In this section we will discuss the two kinds of expressions you may construct:

- Simple
- Complex

as well as how to construct a function.

As we have stated before, an expression is actually data. This is because once BASIC performs all the operations, it returns one data item. An expression may be string or numeric. It may be composed of:

- Constants
- Variables
- Operators
- Functions

Expressions may be either simple or complex:

A **simple expression** consists of a single term: a constant, variable or function. If it is a numeric term, it may be preceded by an optional + or − sign.

For example:

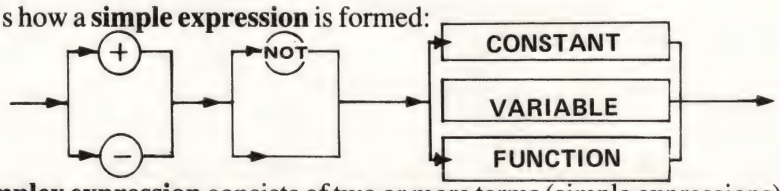
+A 3.3 −5 SQR(8)

are all simple numeric expressions, since they only consist of one numeric term.

A$ STRING$ (20, A$) "WORD" "M"

are all simple string expressions since they only consist of one string term.

Here's how a **simple expression** is formed:



A **complex expression** consists of two or more terms (simple expressions) combined by operators. For example:

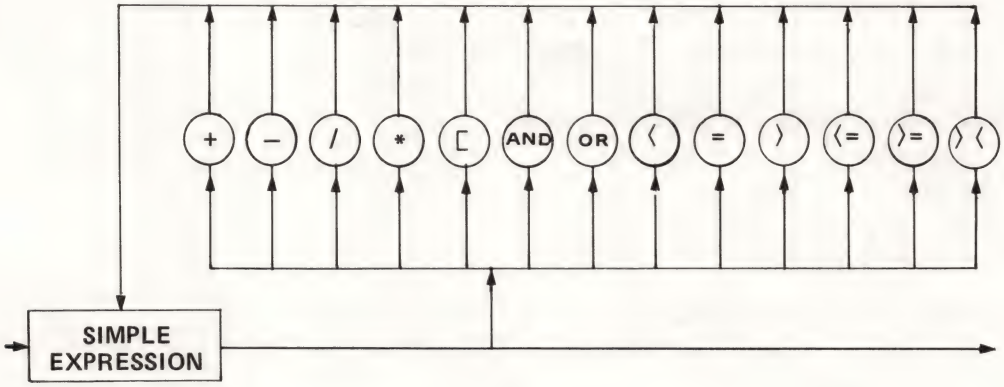
A−1 X+3.2−Y 1=1 A AND B ABS (B)+LOG(2)

are all examples of complex numeric expressions. (Notice that you can use the relational expression (1 = 1) and the logical expression (5 AND 3) as a complex numeric expression since both actually return numeric data.)

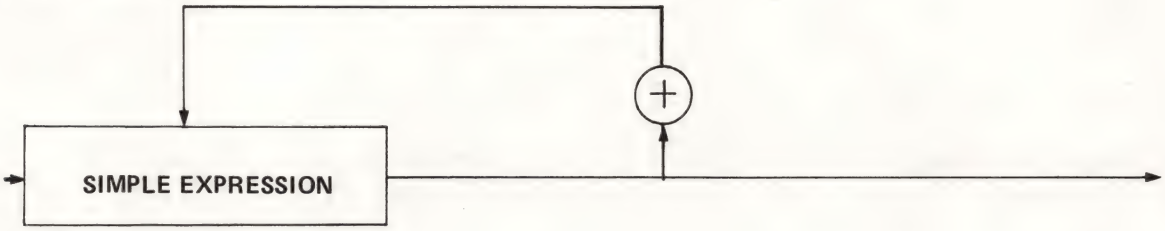
A$+B$ "Z"+Z$ STRING$(10, "A")+ "M"

are all examples of complex string expressions.

This is how a **complex numeric expression** is formed:



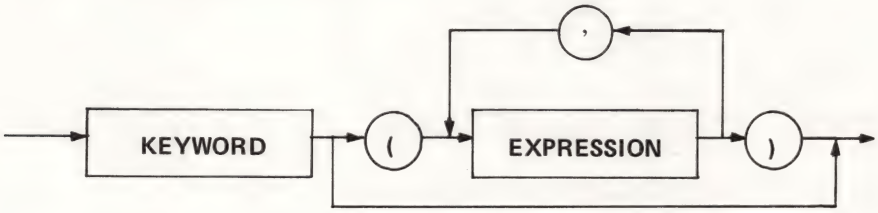
This is how a **complex string expression is formed:**



Most functions, except functions returning system information, require that you input either or both of the following kinds of data:

- One or more numeric expressions
- One or more string expressions.

This is how a **function** is formed:



If the data returned is a number, the function may be used as a term in a numeric expression. If the data is a string, the function may be used as a term in a string expression.

2/Commands

Whenever a prompt > is displayed, your Computer is in the "Immediate" or "Command" Mode. You can type in a command, (ENTER) *it, and the Computer will respond immediately. This chapter describes the commands you'll use to control the Computer — to change modes, begin input and output procedures, alter program memory, etc.* ***All of these commands — except CONT — may also be used inside your program as statements.*** *In some cases this is useful; other times it is just for very specialized applications.*

The commands described in this chapter are:

AUTO	CONT	EDIT	RUN
CLEAR	CSAVE	LIST	SYSTEM
CLOAD	DELETE	LLIST	TROFF
CLOAD?		NEW	TRON

AUTO *line number, increment*

Turns on an automatic line numbering function for convenient entry of programs — all you have to do is enter the actual program statements. You can specify a beginning line number and an increment to be used between line numbers. Or you can simply type AUTO and press (ENTER), in which case line numbering will begin at 10 and use increments of 10. Each time you press (ENTER), the Computer will advance to the next line number.

Examples:	**to use line numbers**
AUTO	10, 20, 30,. . .
AUTO 5, 5	5, 10, 15,. . .
AUTO 100	100, 110, 120,. . .
AUTO 100, 25	100, 125, 150,. . .
AUTO ,10	0, 10, 20,. . .

To turn off the AUTO function, press the (BREAK) key. (Note: When AUTO brings up a line number which is already being used, an asterisk will appear beside the line number. If you do not wish to re-program the line, press the (BREAK) key to turn off AUTO function.)

CLEAR *n*

When used without an argument (e.g., type CLEAR and press (ENTER)), this command resets all numeric variables to zero, and all string variables to null. When used with an argument (e.g., CLEAR 100), this command performs a second function in addition to the one just described: it makes the specified number of bytes available for string storage.

Example: CLEAR 100 makes 100 bytes available for strings. When you turn on the Computer a CLEAR 50 is executed automatically.

CLOAD *''file name''*

Lets you load a BASIC program stored on cassette. Place recorder/player in Play mode (be sure the proper connections are made and cassette tape has been re-wound to proper position). The file name may be any single character except the double-quote (").

Note: See "Using the Cassette Interface" in the Operation Section for instructions on which baud rate to use.

Entering CLOAD will turn on the cassette machine and load the first program encountered. BASIC also lets you specify a desired "file" in your CLOAD command. For example, CLOAD "A" will cause the Computer to ignore programs on the cassette until it comes to one labeled "A". So no matter where file "A" is located on the tape, you can start at the beginning of the tape; file "A" will be picked out of all the files on the tape and loaded. As the Computer is searching for file "A", the names of the files encountered will appear in the upper right corner of the Display, along with a blinking "*".

Only the first character of the file name is used by the Computer for CLOAD, CLOAD?, and CSAVE operations.

Loading a program from tape automatically clears out the previously stored program. See also **CSAVE.**

CLOAD? *"file name"*

Lets you compare a program stored on cassette with one presently in the Computer. This is useful when you have saved a program onto tape (using CSAVE) and you wish to check that the transfer was successful. You may specify CLOAD? *"file-name"*. If you don't specify a file-name, the first program encountered will be tested. During CLOAD?, the program on tape and the program in memory are compared byte for byte. If there are any discrepancies (indicating a bad dump), the message "BAD" will be displayed. In this case, you should CSAVE the program again. (CLOAD?, unlike CLOAD, does not erase the program memory.)

Be sure to type the question mark or the Computer will interpret your command as CLOAD.

CONT

When program execution has been stopped (by the **BREAK** key or by a STOP statement in the program), type CONT and **ENTER** to continue execution at the point where the stop or break occurred. During such a break or stop in execution, you may examine variable values (using PRINT) or change these values. Then type CONT and **ENTER** and execution will continue with the current variable values. CONT, when used with STOP and the **BREAK** key, is primarily a debugging tool.

NOTE: You cannot use CONT after EDITing your program lines or otherwise changing your program. CONT is also invalid after execution has ended normally. See also **STOP.**

CSAVE *"file name"*

Stores the resident program on cassette tape. (Cassette recorder must be properly connected, cassette loaded, and in the Record mode, before you enter the CSAVE command.) You must specify a file-name with this command. This file-name may be any alpha-numeric character other than double-quote ("). The program stored on tape will then bear the specified file-name, so that it can be located by a CLOAD command which asks for that particular file-name. You should always write the appropriate file-names on the cassette case for later reference.

Examples:

CSAVE "1"	saves resident program and attaches label "1"
CSAVE"A"	saves resident program and attaches label "A"

See also **CLOAD.** and "Using the Cassette Interface" in the Operation Section.

DELETE *line number-line number*

Erases program lines from memory. You may specify an individual line or a sequence of lines, as follows:

DELETE *line number*	Erases one line as specified
DELETE *line number-line number*	Erases all program lines starting with first line number specified and ending with last number specified
DELETE-*line number*	Erases all program lines up to and including the specified number

The upper line number to be deleted must be a currently used number.

Examples:

DELETE 5	Erases line 5 from memory (error if line 5 not used)
DELETE 11-18	Erases lines 11, 18 and every line in between

If you have just entered or edited a line, you may delete that line simply by entering DELETE. (use a period instead of the line number).

EDIT *line number*

Puts the Computer in the Edit Mode so you can modify your resident program. The longer and more complex your programs are, the more important EDIT will be. The Edit Mode has its own selection of subcommands, and we have devoted Chapter 9 to the subject.

LIST *line number-line number*

Instructs the Computer to display all programs lines presently stored in memory. If you enter LIST without an argument, the entire program will scroll continuously up the screen. To stop the automatic scrolling, press (SHIFT) and @ simultaneously. This will freeze the display. Press any key to release the "pause" and continue the automatic scrolling.

To examine one line at a time, specify the desired line number as an argument in the LIST command. To examine a certain sequence of program lines, specify the first and last lines you wish to examine.

Examples:

LIST 50	Displays line 50
LIST 50-150	Displays line 50, 150 and everything in between
LIST 50 –	Displays line 50 and all higher-numbered lines
LIST.	Displays current line (line just entered or edited)
LIST – 50	Displays all lines up to and including line 50

LLIST

Works like LIST, but outputs to the Printer

LLIST	Lists current program to printer.
LLIST 100 –	Lists line 100 to the end of the program to the line printer.
LLIST 100-200	Lists line 100 through 200 to the line printer.
LLIST.	Lists current line to the line printer.
LLIST – 100	Lists all lines up to and including line 100 to the line printer.

See LIST.

NEW

Erases all program lines, sets numeric variables to zero and string variables to null. It does not change the string space allocated by a previous CLEAR *number* statement.

NEW is used in the following program to provide password protection.

```
10 INPUT A$: IF A$ <> "E" THEN 65520
20 REM
30 REM          REST OF PROGRAM HERE
40 REM
65519 END
65520 NEW
```

You can't run the rest of the program until you enter the correct password, in this case an E.

RUN *line number*

Causes Computer to execute the program stored in memory. If no line number is specified, execution begins with lowest numbered program line. If a line number is specified, execution begins with the line number. (Error occurs if you specify an unused line number.) Whenever RUN is executed, Computer also executes a CLEAR.

Examples:

RUN	Execution begins at lowest-numbered line
RUN 100	Execution begins at line 100

RUN may be used inside a program as a statement; it is a convenient way of starting over with a clean slate for continuous-loop programs such as games.

To execute a program without CLEARing variables, use GOTO.

SYSTEM

Puts the Computer in the System Mode, which allows you to load object files (machine-language routines or data). Radio Shack offers several machine-language software packages, such as the Editor-Assembler. You can also create your own object files using the TRS-80 Editor/Assembler.

To load an object file: Type SYSTEM and (ENTER)

*?

will be displayed. Now enter the file name (no quotes are necessary) and the tape will begin loading. During the tape load, the familiar asterisks will flash in the upper right-hand corner of the Video Display. When loading is complete, another

*?

will be displayed. Type in a slash-symbol / followed by the address (in decimal form) at which you wish execution to begin. Or you may simply type in the slash-symbol and (ENTER) without any address. In this case execution will begin at the address specified by the object file.

NOTE: BASIC object files are stored as blocks. Further, each block has its own check sum. Should a check sum error occur while loading, the leftmost asterisk will change into the letter C. If this occurs you will have to reload the entire object file. (If the tape motion doesn't stop, hold down (BREAK) until READY returns.)

See "Using the Cassette Interface" in the Operation Section for information on which baud rate to use and the procedures for loading a system tape.

TROFF

Turns off the Trace function. See **TRON.**

TRON

Turns on a Trace function that lets you follow program-flow for debugging and execution analysis. Each time the program advances to a new program line, that line number will be displayed inside a pair of brackets.

For example, enter the following program:

```
10 PRINT "LINE 10"
20 INPUT "PRESS <ENTER> TO BEGIN THE LOOP"; X
30 PRINT "HERE WE GO..."
40 GOTO 30
```

Now type in TRON (**ENTER**), and RUN (**ENTER**).

```
<10>LINE 10
<20>PRESS <ENTER> TO BEGIN THE LOOP?
<30>HERE WE GO...
<40><30>HERE WE GO...
<40><30>HERE WE GO...
```

etc.

(Press (**SHIFT**) and @ simultaneously to pause execution and freeze display. Press any key to continue with execution.)
As you can see from the display, the program is in an infinite loop.

The numbers show you exactly what is going on. (To stop execution, press (**BREAK**).)

To turn off the Trace function, enter TROFF. TRON and TROFF may be used inside programs to help you tell when a given line is executed.

For Example

```
50 TRON
60 A = A + 1
70 TROFF
```

might be helpful in pointing out every time line 60 is executed (assuming execution doesn't jump directly to 60 and bypass 50). Each time these three lines are executed, <60> <70> will be displayed. Without TRON, you wouldn't know whether the program was actually executing line 60. After a program is debugged, TRON and TROFF lines can be removed.

3/Input-Output

The statements described in this chapter let you send data from Keyboard to Computer, Computer to Display, and back and forth between Computer and the Cassette and the Line Printer (if you have one). These will primarily be used inside programs to input data and output results and messages.

Statements covered in this chapter:

PRINT

@ (PRINT modifier)
TAB ((PRINT modifier)
USING (PRINT formatter)

INPUT
DATA
READ
RESTORE
LPRINT
PRINT #-1(Output to Cassette)
INPUT #-1(Input to Cassette)

PRINT *item list*

Prints an item or a list of items on the Display. The items may be either string constants (messages enclosed in quotes), string variables, numeric constants (numbers), variables, or expressions involving all of the preceding items. The items to be PRINTed may be separated by commas or semi-colons. If commas are used, the cursor automatically advances to the next print zone before printing the next item. If semi-colons are used, no space is inserted between the items printed on the Display. In cases where no ambiguity would result, all punctuation can be omitted.

Examples:

```
30 X = 5
40 PRINT 25; "IS EQUAL TO"; X ↑ 2
50 END

80 A$ =  "STRING"
90 PRINT A$; A$, A$; " "; A$
100 END

130 X = 25
140 PRINT 25 "IS EQUAL TO" X
150 END

180 A = 5: B = 10: C = 3
190 PRINT ABC
200 END
```

Postive numbers are printed with a leading blank (instead of a plus sign); **all numbers** are printed with a trailing blank; and no blanks are inserted before or after **strings** (you can insert them with quotes as in line 90).

In line 140 no punctuation is needed; but in line 190 zero will print out because ABC is interpreted as a single variable which has not been assigned a value yet.

```
230 PRINT "ZONE 1","ZONE 2","ZONE 3","ZONE 4","ZONE 1 ETC"
240 END
```

There are four 16-character print zones per line.

```
270 PRINT "ZONE 1",,"ZONE 3"
280 END
```

The cursor moves to the next print zone each time a comma is encountered.

```
300 PRINT "PRINT STATEMENT #10";
310 PRINT "PRINT STATEMENT #20"
320 END
```

A trailing semi-colon overrides the cursor-return so that the next PRINT begins where the last one left off (see line 300).

If no trailing punctuation is used with PRINT, the cursor drops down to the beginning of the next line.

PRINT @ *position, item list*

Specifies exactly where printing is to begin. The @ modifier must be a number from 0 to 1023. Refer to the Video Display worksheet, Appendix C, for the exact position of each location 0-1023:

100 PRINT @ 550, "LOCATION 550"

RUN this to find out where location 550 is.

100 PRINT @ 550, 550; @ 650, 650

Whenever you PRINT @ on the bottom line of the Display, there is an automatic line-feed, causing everything displayed to move up one line. To suppress this, use a trailing semi-colon at the end of the statement.

Example:

```
100 PRINT @ 1000, 1000;
110 GOTO 110
```

Use a trailing semi-colon or comma any time you want to suppress the line feed.

PRINT TAB *(expression)*

Moves the cursor to the specified position on the current line (modulo * 128 if you specify TAB positions greater than 127). TAB may be used several times in a PRINT list.

The value of *expression* must be between 0 and 255 inclusive.

Example:

```
10 PRINT TAB (5) "TABBED 5"; TAB(25) "TABBED 25"
```

No punctuation is required after a TAB modifier.

```
340 'FROM PRINT TAB(EXPRESSION)
350 X = 3
369 PRINT TAB(X) X; TAB(X ↑ 2) X ↑ 2; TAB(X ↑ 3) X ↑ 3
370 END
```

Numerical expressions may be used to specify a TAB position. This makes TAB very useful for graphs of mathematical functions, tables, etc. TAB cannot be used to move the cursor to the left. If cursor is beyond the specified position, the TAB is ignored.

***Modulo** A cyclic counting system. Modulo 64 means the count goes from zero to 63 and then starts over at zero.

PRINT USING *string; item list*

This statement allows you to specify a format for printing string and numeric values. It can be used in many applications such as printing report headings, accounting reports, checks, or wherever a specific print format is required.

The PRINT USING statement uses the following format:
PRINT USING *string ; value*

String and *value* may be expressed as variables or constants. This statement will print the expression contained in the string, inserting the numeric value shown to the right of the semicolon as specified by the field specifiers.

The following field specifiers may be used in the string:

This sign specifies the position of each digit located in the numeric value. The number of # signs you use establishes the numeric field. If the numeric field is greater than the number of digits in the numeric value, then the unused field positions to the left of the number will be displayed as spaces and those to the right of the decimal point will be displayed as zeros.

. The decimal point can be placed anywhere in the numeric field established by the # sign. Rounding-off will take place when digits to the right of the decimal point are suppressed.

, The comma — when placed in any position between the first digit and the decimal point — will display a comma to the left of every third digit as required. The comma establishes an additional position in the field.

****** Two asterisks placed at the beginning of the field will cause all unused positions to the left of the decimal to be filled with asterisks. The two asterisks will establish two more positions in the field.

$ A dollar-sign will be printed ahead of the number.

$$ Two dollar signs placed at the beginning of the field will act as a floating dollar sign. That is, it will occupy the first position preceding the number.

****$** If these three signs are used at the beginning of the field, then the vacant positions to the left of the number will be filled by the * sign and the $ sign will again position itself in the first position preceding the number.

↑ ↑ ↑ ↑ or [[[[Causes the number to be printed in exponential (E or D) format. This will be displayed as a "[".

+	When a + sign is placed at the beginning or end of the field, it will be printed as specified as a + for positive numbers or as a − for negative numbers.
−	When a − sign is placed at the end of the field, it will cause a negative sign to appear after all negative numbers and will appear as a space for positive numbers.
% spaces %	To specify a string field of more than one character, *% spaces %* is used. The length of the string field will be 2 plus the number of spaces between the percent signs.
!	Causes the Computer to use the first string character of the current value.

Any other character that you include in the USING string will be displayed as a string literal.

The following program will help demonstrate these format specifiers:

```
10 INPUT "TYPE IN FORMAT, THEN DATA"; A$, A
20 PRINT USING A$; A
30 GOTO 10
```

RUN this program and try various specifiers and strings for A$ and various values for A.

For Example:

```
>RUN
TYPE IN FORMAT, THEN DATA? ##.#, 12.12
12.1
TYPE IN FORMAT, THEN DATA? ##.#, 1.34
 1.3
TYPE IN FORMAT, THEN DATA? ###.##, 1000.33
%1000.33
TYPE IN FORMAT, THEN DATA?
```

The % sign is automatically printed if the field is not large enough to contain the number of digits found in the numeric value. The entire number to the left of the decimal will be displayed preceded by this sign.

```
>RUN
TYPE IN FORMAT, THEN DATA? ##.##, 12.127
12.13
TYPE IN FORMAT, THEN DATA?
```

Note that the number was rounded to two decimal places.

```
TYPE IN FORMAT, THEN DATA? +##.##, 12.12
+12.12
TYPE IN FORMAT, THEN DATA? "THE ANSWER IS +##.##", -12.12
THE ANSWER IS -12.12
TYPE IN FORMAT, THEN DATA? ##.##+, 12.12
12.12+
TYPE IN FORMAT, THEN DATA? ##.##+, -12.12
12.12-
TYPE IN FORMAT, THEN DATA? ##.##-, 12.12
12.12
TYPE IN FORMAT, THEN DATA? ##.##-, -12.12
12.12-

TYPE IN FORMAT, THEN DATA? "**## IN TOTAL.", 12.12
**12 IN TOTAL.
TYPE IN FORMAT, THEN DATA? $###.##, 12.12
$ 12.12
TYPE IN FORMAT, THEN DATA? $$###.##, 12.12
  $12.12
TYPE IN FORMAT, THEN DATA? **$###.##, 12.12
***$12.12
TYPE IN FORMAT, THEN DATA? "#,###,###", 1234567
1,234,570
TYPE IN FORMAT, THEN DATA?
```

Another way of using the PRINT USING statement is with the string field specifiers "!" and % spaces %.

Examples:

PRINT USING "!"; *string*
PRINT USING "% %"; *string*

The "!" sign will allow only the first letter of the string to be printed. The "% *spaces* %" allows spaces + 2 characters to be printed. Again, the *string* and specifier can be expressed as string variables. The following program will demonstrate this feature:

```
10 INPUT "TYPE IN THE FORMAT, THEN THE STRING DATA"; A$, B$
20 PRINT USING A$; B$
30 GOTO 10
```

and RUN it:

```
TYPE IN THE FORMAT, THEN THE STRING DATA? !, ABCDE
A
TYPE IN THE FORMAT, THEN THE STRING DATA? %%, ABCDE
AB
TYPE IN THE FORMAT, THEN THE STRING DATA? %  %, ABCDE
ABCD
TYPE IN THE FORMAT, THEN THE STRING DATA?
```

Multiple strings or string variables can be joined together (concatenated) by these specifiers. The "!" sign will allow only the first letter of each string to be printed. For example:

```
10 INPUT "TYPE IN THREE NAMES"; A$, B$, C$
20 PRINT USING "!"; A$, B$, C$
30 GOTO 10
```

And RUN it. . .

```
>RUN
TYPE IN THREE NAMES? ABC, DEF, GHI
ADG
TYPE IN THREE NAMES?
```

By using more than one "!" sign, the first letter of each string will be printed with spaces inserted corresponding to the spaces inserted between the "!" signs. To illustrate this feature, make the following change to the last little program:

```
20 PRINT USING "! ! !"; A$, B$, C$
```

And RUN it. . .

```
>RUN
TYPE IN THREE NAMES? ABC, DEF, GHI
A D G
TYPE IN THREE NAMES?
```

Spaces now appear between letters A, D and G to correspond with those placed between the three "!" signs.

Try changing "!!!" to "%%" in line 20 and run the program.

The following program demonstrates one possible use for the PRINT USING statement.

```
510 CLS
520 A$ = "**$##,######.## DOLLARS"
530 INPUT "WHAT IS YOUR FIRST NAME";  F$
540 INPUT "WHAT IS YOUR MIDDLE NAME"; M$
550 INPUT "WHAT IS YOUR LAST NAME"; L$
560 INPUT "ENTER THE AMOUNT PAYABLE"; P
570 PRINT:  PRINT "PAY TO THE ORDER OF ";
580 PRINT USING "!. !. %                %"; F$, M$, L$
600 PRINT: PRINT USING A$; P
620 END
```

RUN the program. Remember, to save programming time, use the "?" sign for PRINT. Your display should look something like this:

```
WHAT IS YOUR FIRST NAME? ALBERT
WHAT IS YOUR MIDDLE NAME? BARCUSSI
WHAT IS YOUR LAST NAME? COOSEY
ENTER THE AMOUNT PAYABLE? 12385.34

PAY TO THE ORDER OF A. B. COOSEY

*****$12,385.30 DOLLARS
```

If you want to use a double-precision amount without rounding off or going into scientific notation, then simply add the double precision sign (#) after the variable P in Lines 560 and 600. You will then be able to use amounts up to 16 decimal places long.

INPUT *item list*

Causes Computer to stop execution until you enter the specified number of values via the keyboard. The INPUT statement may specify a list of string or numeric variables to be input. The items in the list must be separated by commas.

100 INPUT X$, X1, Z$, Z1

This statement calls for you to input a string-literal, a number, another string literal, and another number, **in that order.** When the statement is encountered, the Computer will display a

?

You may then enter the values all at once or one at a time. To enter values all at once, separate them by commas. (If your string literal includes leading blanks, colons, or commas, you must enclose the string in quotes.)

For example, when line 100 (above) is RUN and the Computer is waiting for your input, you could type

JIM,50,JACK,40 (ENTER)

The Computer will assign values as follows:

X$ = "JIM" X1 = 50 Z$ = "JACK" Z1 = 40

If you (ENTER) the values one at a time, the Computer will display a

??

. . . indicating that more data is expected. Continue entering data until all the variables have been set, at which time the Computer will advance to the next statement in your program.

Be sure to enter the correct type of value according to what is called for by the INPUT statement. For example, you can't input a string-value into a numerical variable. If you try to, the Computer will display a

```
?REDO
?
```

and give you another chance to enter the correct type of data value, starting with the *first* value called for by the INPUT list. The Computer *will* accept numeric data for string input.

NOTE: You cannot input an expression into a numerical value — you must input a simple numerical constant.

Example:

```
10 INPUT X1, Y1$
20 PRINT X1, Y1$
30 END
>RUN
? 7 + 3
?REDO
? 10
?? "THIS IS A COMMA  ,  "
 10              THIS IS A COMMA  ,
```

It was necessary to put quotes around "THIS IS A COMMA," because the string contained a comma.

If you type in more data elements than the INPUT statement specifies, the Computer will display the message

```
?EXTRA IGNORED
```

and continue with normal execution of your program.

If you press **ENTER** without typing anything, the variables will have the values they were previously assigned.

You can also include a "prompting message" in your INPUT statement. This will make it easier to input the data correctly. The prompting message must immediately follow "INPUT", must be enclosed in quotes, and must be followed by a semi-colon.

Example:

```
10 INPUT "ENTER NAME, AGE"; N$, A
20 PRINT "HELLO, "; N$; ", YOU ARE AT LEAST"; A * 365; "DAYS OLD"

RUN
ENTER NAME, AGE? DO RAMEY, 31
HELLO, DO RAMEY, YOU ARE AT LEAST 11315 DAYS OLD
```

DATA *item list*

Lets you store data inside your program to be accessed by READ statements. The data items will be read sequentially, starting with the first item in the first DATA statement, and ending with the last item in the last DATA statement. Items in a DATA list may be string or numeric constants — no expressions are allowed. If your string values include colons, commas or leading blanks, you must enclose these values in quotes.

It is important that the data types in a DATA statement match up with the variable types in the corresponding READ statement. DATA statements may appear anywhere it is convenient in a program. Generally, they are placed consecutively, but this is not required.

Examples:

```
10 READ N1$, N2$, N3, N4
20 DATA THIS IS ITEM ONE, THIS IS ITEM TWO, 3, 4
30 PRINT N1$, N2$, N3, N4
```

See **READ, RESTORE.**

READ *item list*

Instructs the Computer to read a value from a DATA statement and assign that value to the specified variable. The first time a READ is executed, the first value in the first DATA statement will be used; the second time, the second value in the DATA statement will be read. When all the items in the first DATA statement have been read, the next READ will use the first value in the second DATA statement; etc. (An Out-of-Data error occurs if there are more attempts to READ than there are DATA items.) The following program illustrates a common application for READ/DATA statements.

```
700 PRINT "NAME","AGE"
710 READ N$
720 IF N$ = "END" THEN PRINT "END OF LIST": END
730 READ AGE
740 IF AGE < 18 PRINT N$, AGE
750 GOTO 710
760 DATA "SMITH, JOHN", 30, "ANDERSON,T.M.", 20
770 DATA "JONES, BILL", 15, "DOE, SALLY", 21
780 DATA "COLLINS, ANDY", 17, END
```

The program locates and prints all the minors' names from the data supplied. Note the use of an END string to allow READing lists of unknown length.

See **DATA, RESTORE**

RESTORE

Causes the next READ statement executed to start over with the first item in the first DATA statement. This lets your program re-use the same DATA lines.

Example:

```
810 READ X
820 RESTORE
830 READ Y
840 PRINT X, Y
850 DATA 50, 60
860 END
```

Because of the RESTORE statement, the second READ statement starts over with the first DATA item.

See **READ, DATA**

LPRINT

This command or statement allows you to output information to the Line Printer. For example, LPRINT A will list the value of A to the line printer. LPRINT can also be used with all the options available with PRINT **except PRINT @**.

Examples:

LPRINT *variable or expression* lists the variable or expression to the line printer.

LPRINT USING prints the information to the line printer using the format specified.

LPRINT TAB will move the line printer carriage position to the right as indicated by the TAB expression.

Example:

```
10 LPRINT TAB (5) "NAME" TAB (30) "ADDRESS" STRING$(63,32) "BALANCE"
```

will print NAME at column 5, ADDRESS at column 30, and BALANCE at column 100.

See **PRINT**.

PRINT #-1, *item list*

Prints the values of the specified variables onto cassette tape. (Recorder must be properly connected and set in Record mode when this statement is executed.)

Example:

```
890 A1 = -30.334: B$ = "STRING-VALUE"
900 PRINT #-1, A1, B$, "THAT'S ALL"
910 END
```

This stores the current values of A1 and B$, and also the string-literal ''THAT'S ALL''. The values may be input from tape later using the INPUT #-1 statement. The INPUT #-1 statement must be identical to the PRINT #-1 statement in terms of **number** and **type of items** in the PRINT #-1/INPUT lists. See **INPUT #-1.**

Special Note:

The values represented in *item list* must not exceed 248 characters total; otherwise all characters after the first 248 will be truncated. For example, PRINT #—1, A#, B#, C#, D#, E#, F#, G#, H#, I#, J#, A$ will probably exceed the maximum record length if A$ is longer than about 75 characters. If you have a lengthy list, you should break it up into two or more PRINT# statements.

INPUT #-1, *item list*

Inputs the specified number of values stored on cassette and assigns them to the specified variable names.

Example:

```
50  INPUT #-1,X,P$,T$
```

When this statement is executed, the Computer will turn on the tape machine, input values in the order specified, then turn off the tape machine and advance to the next statement. If a string is encountered when the INPUT list calls for a number, a bad file data error will occur. If there are not enough data items on the tape to "fill" the INPUT statement, an Out of Data error will occur.

The Input list must be identical to the Print list that created the taped data-block (same number and type of variables in the same sequence.)

Sample Program

Use the two-line program supplied in the **PRINT#** description to create a short data file. Then rewind the tape to the beginning of the data file, make all necessary connections, and put cassette machine in Play mode. Now run the following program.

```
10 INPUT #-1, A1, B$, L$
20 PRINT A1, B$, L$
30 IF L$ = "THAT'S ALL." THEN END
40 REM PROGRAM COULD GO BACK TO LINE 10 FOR MORE DATA
```

This program doesn't care how long or short the data file is, so long as:

1) the file was created by successive PRINT# statements **identical in form** to line 10
2) the last item in the last data triplet is "THAT'S ALL".

4/Program Statements

MODEL III BASIC makes several assumptions about how to run your program. For example:

* *Variables are assumed to be single-precision (unless you use type declaration characters — see Chapter 1, "Variable Types").*
* *A certain amount of memory is automatically set aside for strings and arrays — whether you use all of it or not.*
* *Execution is sequential, starting with the first statement in your program and ending with the last.*

The statements described in this chapter let you override these assumptions, to give your programs much more versatility and power.

***NOTE:** All BASIC statements except INPUT and INPUT#-1 can be used in the Immediate Mode as well as in the Execute Mode.*

Statements described in this chapter:

Type Definition	Assignment & Allocation	Sequence of Execution	Tests (Conditional Statements)
DEFINT	CLEAR *n*	END	IF
DEFSNG	DIM	STOP	THEN
DEFDBL	LET	GOTO	ELSE
DEFSTR		GOSUB	
		RETURN	
		ON. . . GOTO	
		ON. . . GOSUB	
		FOR-NEXT-STEP	
		ERROR	
		ON ERROR GOTO	
		RESUME	
		REM	

DEFINT *letter range*

Variables beginning with any letter in the specified range will be stored and treated as integers, unless a type declaration character is added to the variable name. This lets you conserve memory, since integer values take up less memory than other numeric types. And integer arithmetic is faster than single or double precision arithmetic. However, a variable defined as integer can only take on values between −32768 and +32767 inclusive.

Examples:

```
10  DEFINT A,    I,    N
```

After line 10, all variables beginning with A, I or N will be treated as integers. For example, A1, AA, I3 and NN will be integer variables. However, A1#, AA#, I3# would still be double precision variables, because of the type declaration characters, which always over-ride DEF statements.

```
10  DEFINT I-N
```

Causes variables beginning with I, J, K, L, M or N to be treated as integer variables.

DEFINT may be placed anywhere in a program, but it may change the meaning of variable references without type declaration characters. Therefore it is normally placed at the beginning of a program.

See DEFSNG, DEFDBL, and Chapter 1.

DEFSNG *letter range*

Causes any variable beginning with a letter in the specified range to be stored and treated as single precision, unless a type declaration character is added. Single precision variables and constants are stored with 7 digits of precision and printed out with 6 digits of precision. Since all numeric variables are assumed to be single precision unless DEFined otherwise, the DEFSNG statement is primarily used to re-define variables which have previously been defined as double precision or integer.

Example:

```
100  DEFSNG    I,    W-Z
```

Causes variables beginning with the letter I or any letter W through Z to be treated as single precision. However, I% would still be an integer variable, and I# a double precision variable, due to the use of type declaration characters.

See DEFINT, DEFDBL, and Chapter 1.

DEFDBL *letter range*

Causes variables beginning with any letter in the specified range to be stored and treated as double-precision, unless a type declaration character is added. Double precision allows 17 digits of precision; 16 digits are displayed when a double precision variable is PRINTed.

Example:

```
10  DEFDBL    S-Z,    A-E
```

Causes variables beginning with one of the letters S through Z or A through E to be double precision.

DEFDBL is normally used at the beginning of a program, because it may change the meaning of variable references without type declaration characters.

See DEFINT, DEFSNG, and Chapter 1.

DEFSTR *letter range*

Causes variables beginning with one of the letters in the specified range to be stored and treated as strings, unless a type declaration character is added. If you have CLEARed enough string storage space, each string can store up to 255 characters.

Example:

```
10  DEFSTR    L-Z
```

Causes variables beginning with any letter L through Z to be string variables, unless a type declaration character is added. After line 10 is executed, the assignment L1 = "WASHINGTON" will be valid.

See **CLEAR** *n*, Chapter 1, and Chapter 5.

CLEAR *n*

When used with an argument *n* (*n* can be a constant or an expression), this statement causes the Computer to set aside *n* bytes for string storage. In addition all variables are set to zero. When the TRS-80 is turned on, 50 bytes are automatically set aside for strings.

The amount of string storage CLEARed must equal or exceed the greatest number of characters stored in string variables during execution; otherwise an Out of String Space error will occur.

Example:

```
10  CLEAR 1000
```

Makes 1000 bytes available for string storage.

By setting string storage to the exact amount needed, your program can make more efficient use of memory. A program which uses no string variables could include a CLEAR 0 statement, for example. The CLEAR argument must be non-negative, or an error will result.

DIM *name (dim1, dim2, . . ., dimK)*

Lets you set the "depth" (number of elements allowed per dimension) of an array or list of arrays. If no DIM statement is used, a depth of 11 (subscripts 0-10) is allowed for each dimension of each array used. To create an array with more than three dimensions, you must use DIM.

Example:

```
10  DIM     A(5),     B(2,3),     C$(20)
```

Sets up a one-dimension array A with subscripted elements 0-5; a two-dimension array B with subscripted elements 0,0 to 2,3; and a one-dimension string array C$ with subscripted elements 0-20. Unless previously defined otherwise, arrays A and B will contain single-precision values.

DIM statements may be placed anywhere in your program, and the depth specifier may be a number or a numerical expression.

Example:

```
40  INPUT     "NUMBER OF NAMES";     N
50  DIM     NA (N,2)
```

To re-dimension an array, you must first use a CLEAR statement, either with or without an argument. Otherwise an error will result.

Example Program:

```
10 AA(4) = 11.5
20 DIM AA(7)
READY
>RUN
?DD ERROR IN 20
READY
```

See Chapter 6, ARRAYS.

LET *variable = expression*

May be used when assigning values to variables. Radio Shack Model III BASIC does not require LET with assignment statements, but you might want to use it to ensure compatibility with those versions of BASIC that do require it.

Examples:

```
100  LET A$ = "A ROSE IS A ROSE"
110  LET B1 = 1.23
120  LET X = X - Z1
```

In each case, the variable on the left side of the equals sign is assigned the value of the constant or expression on the right side.

END

Terminates execution normally (without a BREAK message). Some versions of BASIC require END as the last statement in a program; with Model III BASIC it is optional. END is primarily used to force execution to terminate at some point other than the physical end of the program.

Example:

```
10 INPUT S1, S2
20 GOSUB 100
30 REM            MORE PROGRAM LINES HERE...
99 END              : REM        PROTECTIVE END-BLOCK
100 H = SQR(S1*S1 + S2*S2)
110 RETURN
```

The END statement in line 99 prevents program control from "crashing" into the subroutine. Now line 100 can only be accessed by a branching statement such as 20 GOSUB 100.

STOP

Interrupts execution and prints a BREAK IN *line number* message. STOP is primarily a debugging aid. During the break in execution, you can examine or change variable values. The command CONT can then be used to re-start execution at the point where it left off. (If the program itself is altered during a break, CONT cannot be used.)

Example:

```
10 X = RND(10)
20 STOP
30 GOSUB 1000
99 END
1000 REM
1010 RETURN
```

Suppose we want to examine what value for X is being passed to the subroutine beginning at line 1000. During the break, we can examine X with PRINT X.

GOTO *line number*

Transfers program control to the specified line number. Used alone, GOTO *line number* results in an unconditional (or automatic) branch; however, test statements may precede the GOTO to effect a conditional branch.

Example:

```
200 GOTO 10
```

When 200 is executed, control will automatically jump back to line 10.

You can use GOTO in the Immediate Mode as an alternative to RUN. GOTO *line number* causes execution to begin at the specified line number, **without an automatic CLEAR.** This lets you pass values assigned in the Immediate Mode to variables in the Execute Mode.

See **IF,THEN,ELSE,ON... GOTO.**

GOSUB *line number*

Tranfers program control to the subroutine beginning at the specified line number and stores an address to RETURN to after the subroutine is complete. When the Computer encounters a RETURN statement in the subroutine, it will then return control to the statement which follows GOSUB.

If you don't RETURN, the previously stored address will not be deleted from the area of memory used for saving information, called the stack. The stack might eventually overflow, but, even more importantly, this address might be read incorrectly during another operation, causing a hard-to-find program error. So. . . always RETURN from your subroutines. GOSUB, like GOTO may be preceded by a test statement. See **IF,THEN,ELSE,ON...GOSUB.**

Example Program:

```
100 GOSUB 200
110 PRINT "BACK FROM THE SUBROUTINE": END
200 PRINT "EXECUTING THE SUBROUTINE"
210 RETURN
READY
>RUN
EXECUTING THE SUBROUTINE
BACK FROM THE SUBROUTINE
```

Control branches from line 100 to the subroutine beginning at line 200. Line 210 instructs Computer to return to the statement immediately following GOSUB, that is, line 110.

RETURN

Ends a subroutine and returns control to statement immediately following the most recently executed GOSUB. If RETURN is encountered without execution of a matching GOSUB, an error will occur. See **GOSUB.**

ON *n* GOTO *line number, ..., line number*

This is a multi-way branching statement that is controlled by a test variable or expression. The general format for ON *n* GOTO is:

ON *expression* GOTO *1st line number, 2nd line number,. . ., Kth line number*

expression must be between 0 and 255 inclusive.

When ON. . . GOTO is executed, first the expression is evaluated and the integer portion. . . INT(expression). . . is obtained. We'll refer to this integer portion as J. The Computer counts over to the Jth element in the line-number list, and then branches to the line number specified by that element. If there is no Jth element (that is, if J > K or J = 0 in the general format above), then control passes to the next statement in the program.

If the test expression or number is less than zero, or greater than 255, an error will occur. The line-number list may contain any number of items.

For example:

100 ON MI GOTO 150, 160, 170, 150, 180

says "Evaluate MI. If integer portion of MI equals 1 then go to line 150;
If it equals 2, then go to 160;
If it equals 3, then go to 170;
If it equals 4, then go to 150;
If it equals 5, then go to 180;
If the integer portion of MI doesn't equal any of the numbers 1 through 5, advance to the next statement in the program."

Sample Program

```
100 INPUT "ENTER A NUMBER"; X
110 ON SGN(X) + 2 GOTO 200, 210, 220
200 PRINT "NEGATIVE": END
210 PRINT "ZERO": END
220 PRINT "POSITIVE": END
```

SGN(X) returns − 1 for X less than zero; 0 for X equal to zero; and + 1 for X greater than 0. By adding 2, the expression takes on the values 1, 2, and 3, depending on whether X is negative, zero, or positive. Control then branches to the appropriate line number.

ON *n* GOSUB *line number, ..., line number*

Works like ON *n* GOTO, except control branches to one of the subroutines specified by the line numbers in the line-number list.

Example:

```
100 INPUT "CHOOSE 1, 2, OR 3"; I
110 ON I GOSUB 200, 300, 400
120 END
200 PRINT "SUBROUTINE #1": RETURN
300 PRINT "SUBROUTINE #2": RETURN
400 PRINT "SUBROUTINE #3": RETURN
```

The test object *n* may be a numerical constant, variable or expression. It must have a non-negative value or an error will occur.

See **ON *n* GOTO.**

FOR *counter* = *exp* TO *exp* STEP *exp*
NEXT *counter*

Opens an iterative (repetitive) loop so that a sequence of program statements may be executed over and over a specified number of times. The general form is (brackets indicate optional material):

line # FOR *counter-variable* = *initial value* TO *final value* [STEP *increment*]

.

.

.

line # NEXT [*counter-variable*]

In the FOR statement, *initial value, final value* and *increment* can be constants, variables or expressions. The first time the FOR statement is executed, these three are evaluated and the values are saved; if the variables are changed by the loop, it will have no effect on the loop's operation. However, **the counter variable must not be changed** or the loop will not operate normally.

The FOR-NEXT-STEP loop works as follows: the first time the FOR statement is executed, the counter is set to the "initial value." Execution proceeds until a NEXT statement is encountered. At this point, the counter is incremented by the amount specified in the STEP *increment*. (If the *increment* has a negative value, then the counter is actually decremented.) If STEP *increment* is not used, an increment of 1 is assumed.

Then the counter is compared with the *final value* specified in the FOR statement. If the counter is greater than the *final value,* the loop is completed and execution continues with the statement following the NEXT statement. (If *increment* was a negative number, loop ends when counter is **less** than *final value*.) If the counter has not yet exceeded the *final value,* control passes to the first statement after the FOR statement.

Example Programs:

```
10 FOR I = 10 TO 1 STEP -1
20 PRINT I;
30 NEXT
READY
>RUN
 10  9  8  7  6  5  4  3  2  1
READY
>
10 FOR K = 0 TO 1 STEP .3
20 PRINT K;
30 NEXT
READY
>RUN
 0  .3  .6  .9
READY
>
```

After K = .9 is incremented by .3, K = 1.2. This is greater than the *final value* 1, therefore loop ends without ever printing *final value*.

```
10 FOR K = 4 TO 0
20 PRINT K;
30 NEXT
READY
>RUN
 4
READY
>
```

No STEP is specified, so STEP 1 is assumed. After K is incremented the first time, its value is 5. Since 5 is greater than the *final value* 0, the loop ends.

```
10 J = 3: K = 8: L = 2
20 FOR I = J TO K + 1 STEP L
30 J = 0: K = 0: L = 0
40 PRINT I;
50 NEXT
READY
>RUN
 3  5  7  9
READY
>
```

The variables and expressions in line 20 are evaluated once and these values become constants for the FOR-NEXT-STEP loop. Changing the variable values later has no effect on the loop.

FOR-NEXT loops may be ''nested'':

```
10 FOR I = 1 TO 3
20   PRINT "OUTER LOOP"
30      FOR J = 1 TO 2
40        PRINT "  INNER LOOP"
50      NEXT J
60 NEXT I
```

```
RUN
OUTER LOOP
  INNER LOOP
  INNER LOOP
OUTER LOOP
  INNER LOOP
  INNER LOOP
OUTER LOOP
  INNER LOOP
  INNER LOOP
READY
>
```

Note that each NEXT statement specifies the appropriate counter variable; however, this is just a programmer's convenience to help keep track of the nesting order. The counter variable may be omitted from the NEXT statements. But if you **do** use the counter variables, you **must** use them in the right order; i.e., the counter variable for the innermost loop must come first.

It is also advisable to specify the counter variable with NEXT statements when your program allows branching to program lines outside the FOR-NEXT loop.

Another option with nested NEXT statements is to use a counter variable list.

Delete line 50 from the above program and change line 60:

```
60  NEXT J,I
```

Loops may be nested 3-deep, 4-deep, etc. The only limit is the amount of memory available.

ERROR *code*

Lets you "simulate" a specified error during program execution. The major use of this statement is for testing an ON ERROR GOTO routine. When the ERROR *code* statement is encountered, the Computer will proceed exactly as if that kind of error had occurred. Refer to Appendix B for a listing of error codes and their meanings.

Example Program:

```
100 ERROR 1
READY
>RUN
?NF Error in 100
READY
>
```

1 is the error code for "attempt to execute NEXT statement without a matching FOR statement".

See **ON ERROR GOTO, RESUME.**

ON ERROR GOTO *line number*

When the Computer encounters any kind of error in your program, it normally breaks out of execution and prints an error message. With ON ERROR GOTO, you can set up an error-trapping routine which will allow your program to "recover" from an error and continue, without any break in execution. Normally you have a particular type of error in mind when you use the ON ERROR GOTO statement. For example, suppose your program performs some division operations and you have not ruled out the possibility of division by zero. You might want to write a routine to handle a division-by-zero error, and then use ON ERROR GOTO to branch to that routine when such an error occurs.

Example:

```
10 ON ERROR GOTO 100
20 A = 1 / 0
90 END
100 PRINT"ERROR # "; ERR/2 + 1
110 RESUME 90
```

In this "loaded" example, when the Computer attempts to execute line 20, a divide-by-zero error will occur. But because of line 10, the Computer will simply ignore line 20 and branch to the error-handling routine beginning at line 100.

NOTE: The ON ERROR GOTO must be executed **before** the error occurs or it will have no effect.

The ON ERROR GOTO statement can be disabled by executing an ON ERROR GOTO 0. If you use this inside an error-trapping routine, BASIC will handle the current error normally.

The error handling routine must be terminated by a RESUME statement. See **RESUME.**

RESUME *line number*

Terminates an error handling routine by specifying where normal execution is to resume.

RESUME without a line number and RESUME 0 cause the Computer to return to the statement in which the error occurred.

RESUME followed by a line number causes the Computer to branch to the specified line number.

RESUME NEXT causes the Computer to branch to the statement **following** the point at which the error occurred.

Sample Program with an Error Handling Routine

```
605 ON ERROR GOTO 640
610 INPUT "SEEKING SQUARE ROOT OF"; X
620 PRINT SQR(X)
630 GOTO 610
640 PRINT "IMAGINARY ROOT:"; SQR(-X); " * I"
650 RESUME 610
660 END
```

RUN the program and try inputting a negative value.

You must place a RESUME statement at the end of your error trapping routine, so that later errors may also be trapped.

REM

Instructs the Computer to ignore the rest of the program line. This allows you to insert comments (REMarks) into your program for documentation. Then, when you (or someone else) look at a listing of your program, it'll be a lot easier to figure out. If REM is used in a multi-statement program line, it must be the last statement.

Example Program:

```
710 REM ** THIS REMARK INTRODUCES THE PROGRAM **
720 REM ** AND POSSIBLY THE PROGRAMMER, TOO.  **
730 REM **                                     **
740 REM ** THIS REMARK EXPLAINS WHAT THE      **
750 REM ** VARIOUS VARIABLES REPRESENT:       **
760 REM ** C = CIRCUMFERENCE  R = RADIUS      **
770 REM ** D = DIAMETER                       **
780 REM
```

Any alphanumeric character may be included in a REM statement, and the maximum length is the same as that of other statements: 255 characters total.

In Model III BASIC, an apostrophe ' (SHIFT 7) may be used as an abbreviation for :REM.

```
100 A=1     'THIS, TOO IS A REMARK
```

IF *true/false expression* THEN *action-clause*

Instructs the Computer to test the following logical or relational expression. If the expression is True, control will proceed to the "action" clause immediately following the expression. If the expression is False, control will jump to the matching ELSE statement (if there is one) or down to the next program line.

In numerical terms, if the expression has a non-zero value, it is always equivalent to a logical True.

Examples:

```
100 IF X > 127 THEN PRINT "OUT OF RANGE": END
```

If X is greater than 127, control will pass to the PRINT statement and then to the END statement. But if X is **not** greater than 127, control will jump down to the next line in the program, skipping the PRINT and END statements.

```
IF 0 <= X AND X <= Y THEN Y = X + 180
```

If both expressions are True then Y will be assigned the value X + 180. Otherwise control will pass directly to the next program line, skipping the THEN clause.

See **THEN, ELSE.**

THEN *statement or line number*

Initiates the "action clause" of an IF-THEN type statement. THEN is optional except when it is required to eliminate an ambiguity, as in IF A < 0 100. THEN should be used in IF-THEN-ELSE statements.

ELSE *statement or line number*

Used after IF to specify an alternative action in case the IF test fails. (When no ELSE statement is used, control falls through to the next program line after a test fails.)

Examples:

```
100 INPUT A$: IF A$ = "YES" THEN 300 ELSE END
```

In line 100, if A$ equals "YES" then the program branches to line 300. But if A$ does not equal "YES", program skips over to the ELSE statement which then instructs the Computer to end execution.

```
200 IF A < B THEN PRINT "A<B" ELSE PRINT "B<=A"
```

If A is less than B, the Computer prints that fact, and then proceeds down to the next program line, **skipping** the ELSE statement. If A is not less than B, Computer jumps directly to the ELSE statement and prints the specified message. **Then** control passes to the next statement in the program.

```
200 IF A>.001 THEN B = 1/A: A = A/5: ELSE 260
```

If A>.001 is True, then the next two statements will be executed, assigning new values to B and A. Then the program will drop down to the next line, skipping the ELSE statement. But if A>.001 is False, the program jumps directly over to the ELSE statement, which then instructs it to branch to line 260. Note that GOTO is not required after ELSE.

IF-THEN-ELSE statements may be nested, but you have to take care to match up the IFs and ELSEs.

```
810 INPUT "ENTER TWO NUMBERS"; A, B
820 IF A <= B THEN IF  A <  B  PRINT A;: ELSE PRINT "NEITHER
";: ELSE PRINT B;
830 PRINT "IS SMALLER"
840 END
```

RUN the program, inputting various pairs of numbers. The program picks out and prints the smaller of any two numbers you enter.

5/Strings

"Without string-handling capabilities, a computer is just a super-powered calculator." There's an element of truth in that exaggeration; the more you use the string capabilities of Model III BASIC*, the truer the statement will seem.*

In Model III BASIC *any valid variable name can be used to contain string values, by the* DEFSTR *statement or by adding a type declaration character to the name. And each string can contain up to 255 characters.*

Moreover, you can compare strings to alphabetize them, for example. You can take strings apart and string them together (concatenate them). For background material to this chapter, see Chapter 1, "Variable Types" and "Glossary", and Chapter 4, **DEFSTR.**

Functions covered in this chapter:

FRE *(string)*	LEFT$	STRING$
INKEY$	MID$	TIME$
LEN	RIGHT$	VAL
ASC	STR$	
CHR$		

***NOTE:** Whenever* string *is given as a function argument, you can use a string expression or constant.*

String Space

Fifty bytes of memory are set aside automatically to store strings. If you run out of string space, you will get an OS error and you should use the CLEAR *n* command to save more space.

Note: CLEAR also sets variables to zero or null strings.

To calculate the space you'll need, multiply the amount of space each variable takes (See VARPTR) by the number of string variables you are using, including temporary variables.

Temporary variables are created during the calculation of string functions. Therefore even if you have only a few short string variables assigned in your program, you may run out of string space if you concatenate them several times.

ASC *(string)*

Returns the ASCII code for the first character of the specified string. The string-argument must be enclosed in parentheses. A null-string argument will cause an error to occur.

```
100 PRINT ASC("A")
110 T$ = "AB": PRINT ASC (T$)
```

Lines 100 and 110 will print the same number.

The argument may be an expression involving string operators and functions:

```
200 PRINT ASC(RIGHT$(T$, 1))
```

Refer to the ASCII Code Table, Appendix C.

CHR$ *(expression)*

Performs the inverse of the ASC function: returns a one-character string whose character has the specified ASCII, control or graphics code. The argument may be any number from 0 to 255, or any variable expression with a value in that range. Argument must be enclosed in parentheses.

```
100 PRINT CHR$(35)          Prints a number-sign #
```

Using CHR$, you can even assign quote-marks (normally used as string-delimiters) to strings. The ASCII code for quotes " is 34. So A$ = CHR$(34) assigns the value " to A$.

```
410 A$ = CHR$(34)
420 PRINT "HE SAID, "; A$; "HELLO."; A$
```

CHR$ may also be used to display any of the graphics or special characters. (See Appendix C, Character Codes.)

```
460 CLS
470 FOR I = 129 TO 191
480 PRINT I; CHR$(I),
490 NEXT
500 GOTO 500
```

(RUN the program to see the various graphics characters.)

Codes 0-31 are display control codes. Instead of returning an actual display character, they return a control character. When the control character is PRINTed, the function is performed. For example, 23 is the code for 32 character-per-line format; so the command, PRINT CHR$(23) converts the display format to 32 characters per line. (Hit CLEAR, execute CLS, or execute PRINT CHR$(28) to return to 64 character-per-line format.)

FRE (*string*)

When used with a string variable or string constant as an argument, returns the amount of string storage space currently available. Argument must be enclosed in parentheses. FRE causes BASIC to start searching through memory for unused string space. If your program has done a lot of string processing, it may take several minutes to recover all the "scratch pad" type memory.

```
500 PRINT FRE(A$), FRE(L$), FRE ("Z")
```

All return the same value.

The string used has no significance; it is a dummy variable. See Chapter 4, **CLEAR** *n*.

FRE(*number*) returns the amount of available memory (same as MEM).

INKEY$

Returns a one-character string determined by a keyboard check. The last key pressed before the check is returned. If no key has been pressed, a null string (length zero) is returned. This is a very powerful function because it lets you input values while the Computer is executing — without using the (ENTER) key. The popular video games which let you fire at will, guide a moving dot through a maze, play tennis, etc., may all be simulated using the INKEY$ function (plus a lot of other program logic, of course).

Characters typed to an INKEY$ are **not** automatically displayed on the screen.

INKEY$ is often placed inside some sort of loop, so that the keyboard is scanned repeatedly.

Example Program:

```
540 CLS
550 PRINT @ 540, INKEY$: GOTO 550
```

RUN the program; notice that the screen remains blank until the first time you hit a key. The last key hit remains on the screen until you hit another one. (The last key hit is always saved. The INKEY$ function uses it until it is replaced by a new value.)

INKEY$ may be used in sequences of loops to allow the user to build up a longer string.

Example:

```
590 PRINT "ENTER THREE CHARACTERS"
600 A$ = INKEY$: IF A$ = "" THEN 600 ELSE PRINT A$;
610 B$ = INKEY$: IF B$ = "" THEN 610 ELSE PRINT B$;
620 C$ = INKEY$: IF C$ = "" THEN 620 ELSE PRINT C$;
630 D$ = A$ + B$ + C$
```

A three-character string D$ can now be entered via the keyboard without using the (ENTER) key.

NOTE: The statement IF A$ = " " compares A$ to the null string. There are **no** spaces between the double-quotes.

LEFT$ (*string, n*)

Returns the first *n* characters of *string*. The arguments must be enclosed in parentheses. *string* may be a string constant or expression, and *n* may be a numeric expression.

Example Program:

```
670 A$ = "TIMOTHY"
680 B$ = LEFT$ (A$, 3)
690 PRINT B$; " -THAT'S SHORT FOR "; A$
```

LEN *(string)*

Returns the character length of the specified string. The string variable, expression, or constant must be enclosed in parentheses.

```
730 A$ = ""
740 B$ = "TOM"
750 PRINT A$, B$, B$ + B$
760 PRINT LEN(A$), LEN(B$), LEN(B$+B$)
```

MID$ *(string,p,n)*

Returns a substring of *string* with length *n* and starting at position *p*. The string name, length and starting position must be enclosed in parentheses. *string* may be a string constant or expression, and *n* and *p* may be numeric expressions or constants. For example, MID$ (L$,3,1) refers to a one-character string beginning with the third character of L$.

If no argument is specified for the length *n*, the entire string beginning at position *p* is returned.

Example Program:

The first three digits of a local phone number are sometimes called the "exchange" of the number. This program looks at a complete phone number (area code, exchange, last four digits) and picks out the exchange of that number.

```
800 INPUT "AREA CODE AND NUMBERS (NO HYPHENS, PLEASE)"; P$
810 EX$ = MID$ (P$, 4,3)
820 PRINT "NUMBER IS IN THE "; EX$; " EXCHANGE."
```

RIGHT$ *(string, n)*

Returns the last *n* characters of *string*. *string* and *n* must be enclosed in parentheses. *string* may be a string constant or variable, and n may be a numerical constant or variable. If LEN*(string)* is less than or equal to *n*, the entire *string* is returned.

```
10 INPUT "ENTER A WORD"; M$
20 IF LEN(M$) = 0 THEN 10
30 PRINT "THE LAST LETTER WAS:  "; RIGHT$(M$,1)
40 GOTO 10
```

STR$ *(expression)*

Converts a numeric expression or constant to a string. The numeric expression or constant must be enclosed in parentheses. STR$(A), for example, returns a string equal to the character representation of the value of A. For example, if A = 58.5, then STR$(A) equals the string " 58.5". (Note the leading blank in " 58.5"). While arithmetic operations may be performed on A, only string operations and functions may be performed on the string " 58.5".

PRINT STR$(X) prints X without a trailing blank; PRINT X prints X with a trailing blank.

Example Program:

```
860 A = 58.5: B = -58.5
870 PRINT STR$(A)
880 PRINT STR$(B)
890 PRINT STR$(A+B)
900 PRINT STR$(A) + STR$(B)
```

STRING$ *(n, ''character''* or *number)*

Returns a string composed of *n character*-symbols. For example, STRING$(30,''*'') returns ''**********''. STRING$ is useful in creating graphs, tables, etc.

The argument *n* is any numerical expression with a value of from zero to 255.

character can also be a number from 0-255; in this case, it will be treated as an ASCII, control, or graphics code.

Example:

```
10 CLEAR 200
20 FOR I=128 TO 191
30 A$ = STRING$(64,I)
40 PRINT A$;
50 NEXT I
```

TIME$

Returns today's date and time. Your Model III contains a built-in clock. To use this clock, you will want to first set it to the correct date and time. To do this, you may type and run this little program:

```
10 DEFINT A-Z
20 DIM TM(5)
30 CL = 16924
40 PRINT "INPUT 6 VALUES: MO, DA, YR, HR, MN, SS"
50 INPUT TM(0), TM(1), TM(2), TM(3), TM(4), TM(5)
60 FOR I = 0 TO 5
70        POKE CL - I, TM(I)
80 NEXT I
90 PRINT "CLOCK IS SET"
100 END
```

Once you have set the date and time with this program, you may request it any time you want. For example, this program line:

```
10  PRINT TIME$
```

causes the Computer to print today's date and time.

If you do not set the date and time, the Computer will keep time anyway. However, the date and time will be set at zero when you first turn on the Computer or reset it.

NOTE: The clock is turned off during cassette operations and at certain other times. Therefore it will need to be corrected periodically.

VAL *(string)*

Performs the inverse of the STR$ function: returns the number represented by the characters in a string argument. The numerical type of the result can be integer, single precision, or double precision, as determined by the rules for the typing of constants (See page 1/10 in this section). For example, if A$ = "12" and B$ = "34" then VAL (A$ + "." + B$) returns the value 12.34. VAL(A$ + "E" + B$) returns the value 12E34, that is 12×10^{34}.

VAL operates a little differently on mixed strings — strings whose values consist of a number followed by non-numeric characters. In such cases, only the leading number is used in determining VAL; the non-numeric remainder is ignored.

For example: VAL ("100 DOLLARS") returns 100.

This can be a handy short-cut in examining addresses, for example.

Example Program:

```
940 REM "WHAT SIDE OF STREET?"
950 REM EVEN = NORTH. ODD = SOUTH
960 INPUT "ADDRESS:  NUMBER AND STREET"; AD$
970 C = INT(VAL(AD$)/2) * 2
980 IF C = VAL(AD$) THEN PRINT "NORTH SIDE": GOTO 960
990 PRINT "SOUTH SIDE": GOTO 960
```

RUN the program, entering street addresses like ''1015 SEVENTH AVE''.

If the string is non-numeric or null, VAL returns a zero.

6/Arrays

An array is simply an ordered list of values. In Model III BASIC these values may be either numbers or strings, depending on how the array is defined or typed. Arrays provide a fast and organized way of handling large amounts of data. To illustrate the power of arrays, this chapter traces the development of an array to store checkbook data: check numbers, dates written, and amounts for each check.

In addition, several matrix manipulation subroutines are listed at the end of this chapter. These sequences will let you add, multiply, transpose, and perform other operations on arrays.

***Note:** Throughout this chapter, zero-subscripted elements are generally ignored for the sake of simplicity. But you should remember they are available and should be used for the most efficient use of memory. For example, after DIM A(4), array A contains 5 elements: A(0), A(1), A(2), A(3), A(4).*

*For background information on arrays, see Chapter 4, **DIM**, and Chapter 1, "Arrays".*

A Check-Book Array

Consider the following table of checkbook information:

Check #	Date Written	Amount
025	1-1-78	10.00
026	1-5-78	39.95
027	1-7-78	23.50
028	1-7-78	149.50
029	1-10-78	4.90
030	1-15-78	12.49

Note that every item in the table may be specified simply by reference to two numbers: the row number and the column number. For example, (row 3, column 3) refers to the amount 23.50. Thus the number pair (3,3) may be called the "subscript address" of the value 23.50.

Let's set up an array, CK, to correspond to the checkbook information table. Since the table contains 6 rows and 3 columns, array CK will need two dimensions: one for row numbers, and one for column numbers. We can picture the array like this:

A(1,1) = 025	A(1,2) = 1.0178	A(1.3) = 10.00
.	.	.
.	.	.
.	.	.
.	.	.
A(6,1) = 030	A(6,2) = 1.1578	A(6,3) = 12.49

Notice that the date information is recorded in the form *mm.ddyy*. where *mm* = month number, *dd* = day of month, and *yy* = last two digits of year. **Since CK is a numeric array, we can't store the data with alpha-numeric characters such as dashes.**

Suppose we assign the appropriate values to the array elements. Unless we have used a DIM statement, the Computer will assume that our array requires a depth of 10 for each dimension. That is, the Computer will set aside memory locations to hold CK(7,1), CK (7,2),. . . , CK(10,1), CK(10,2) and CK(10,3). In this case, we don't want to set aside this much space, so we use the DIM statement at the beginning of our program:

```
40 DIM CK(6,3)
```

Now let's add program steps to read the values into the array CK:

```
50 FOR ROW = 1 TO 6
60 FOR COL = 1 TO 3
70 READ CK(ROW,COL)
80 NEXT COL, ROW
90 DATA 025, 1.0178, 10.00
100 DATA 026, 1.0578, 39.95
110 DATA 027, 1.0778, 23.50
120 DATA 028, 1.0778, 149.50
130 DATA 029, 1.1078, 4.90
140 DATA 030, 1.1578, 12.49
```

Now that our array is set up, we can begin taking advantage of its built-in structure. For example, suppose we want to add up all the checks written. Add the following lines to the program:

```
150 FOR ROW = 1 TO 6
160 SUM = SUM + CK(ROW,3)
170 NEXT
180 PRINT "TOTAL OF CHECKS WRITTEN";
190 PRINT USING "$$####.##"; SUM
```

Now let's add program steps to print out all checks that were written on a given day.

```
200 PRINT "SEEKING CHECKS WRITTEN ON WHAT DATE (MM.DD YY)";
210 INPUT DT
220 PRINT: PRINT "ANY CHECKS WRITTEN ARE LISTED BELOW:"
230 PRINT "CHECK #", "AMOUNT": PRINT
240 FOR ROW = 1 TO 6
250 IF CK(ROW,2) = DT THEN PRINT CK(ROW,1), CK(ROW,3)
260 NEXT
```

It's easy to generalize our program to handle checkbook information for all 12 months and for years other than 1978.

All we do is increase the size (or "depth") of each dimension as needed. Let's assume our checkbook includes check numbers 001 through 300, and we want to store the entire checkbook record. Just make these changes:

```
40 DIM CK(300,3)          'SET UP A 300 BY 3 ARRAY
50 FOR ROW = 1 TO 300
```

and add DATA lines for check numbers 001 through 300. You'd probably want to pack more data onto each DATA line than we did in the above DATA lines.

And you'd change all the ROW counter final values:

```
150 FOR ROW = 1 TO 300
240 FOR ROW = 1 TO 300
```

Other Types of Arrays

Remember, in Model III BASIC the number of dimensions an array can have (and the size or depth of the array), is limited only by the amount of memory available. Also remember that **string** arrays can be used. For example, C$(X) would automatically be interpreted as a string array. And if you use DEFSTR A at the beginning of your program, any array whose name begins with A would also be a string array. One obvious application for a string array would be to store text material for access by a string manipulation program.

```
10 CLEAR 1200
20 DIM TXT$(10)
```

would set up a string array capable of storing 10 lines of text. 1200 bytes were CLEARed to allow for 10 sixty-character lines, plus 600 extra bytes for string manipulation with other string variables.

Array/Matrix Manipulation Subroutines

To use this subroutine, your main program must supply values for two variables N1 (number of rows) and N2 (number of columns). Within the subroutine, you can assign values to the elements in the array row by row by answering the INPUT statement.

```
10 FOR ROW = 1 TO N1
20 FOR COL = 1 TO N2
30 PRINT "ENTER DATA FOR "; ROW; ":"; COL
40 INPUT A(ROW,COL)
50 NEXT COL
60 NEXT ROW
70 RETURN
```

To use this subroutine, your main program must supply values for three variables N1 (size of dim #1), N2 (size of dim #2) and N3 (size of dim #3). Within the subroutine, you can assign values to each element of the array using READ and DATA statements. You must supply I x J x K elements in the following order: row by row for K = 1, row by row for K = 2, row by row for K = 3, and so on for each value of N3.

```
400 REM REQUIRES DATA STMTS.
410 FOR K = 1 TO N3
420 FOR I = 1 TO N1
430 FOR J = 1 TO N2
440 READ A(I,J,K)
450 NEXT J, I, K
460 RETURN
```

Main program supplies values for variables N1, N2, N3. The subroutine prints the array.

```
560 FOR K = 1 TO N3
570 FOR I = 1 TO N1
580 FOR J = 1 TO N2
590 PRINT A(I,J,K),
600 NEXT J: PRINT
610 NEXT I: PRINT
620 NEXT K: PRINT
630 RETURN
```

Main program supplies values for variables N1, N2, N3. Within the subroutine, you can assign values to each element of the array using the INPUT statement.

```
660 FOR K = 1 TO N3
670 PRINT "PAGE"; K
680 FOR I = 1 TO N1
690 PRINT "INPUT ROW"; I
700 FOR J = 1 TO N2
710 INPUT A(I,J,K)
720 NEXT J
730  NEXT I
740 PRINT: NEXT K
750 RETURN
```

Multiplication by a Single Variable: Scalar Multiplication (3 Dimensional)

```
780 FOR K = 1 TO N3
790 FOR J = 1 TO N2
800 FOR I = 1 TO N1
810 B(I,J,K) = A(I,J,K) * X
820 NEXT I
830 NEXT J
840 NEXT K
850 RETURN
```

Multiplies each element in MATRIX A by X and constructs matrix B

Transposition of a Matrix (2 Dimensional)

```
880 FOR I = 1 TO N1
890 FOR J = 1 TO N2
900 B(J,I) = A(I,J)
910 NEXT J
920 NEXT I
930 RETURN
```

Transposes matrix A into matrix B

Matrix Addition (3 Dimensional)

```
960 FOR K = 1 TO N3
970 FOR J = 1 TO N2
980 FOR I = 1 TO N1
990 C(I,J,K) = A(I,J,K) + B(I, J, K)
1000 NEXT I
1010 NEXT J
1020 NEXT K
1030 RETURN
```

Array Element-wise Multiplication (3 Dimensional)

```
1060 FOR K = 1 TO N3
1070 FOR J = 1 TO N2
1080 FOR I = 1 TO N1
1090 C(I,J,K) = A(I,J,K) * B(I,J,K)
1100 NEXT I
1110 NEXT J
1120 NEXT K
1130 RETURN
```

Multiplies each element in A times its corresponding element in B.

Matrix Multiplication (2 Dimensional)

```
1160 FOR I = 1 TO N1
1170 FOR J = 1 TO N2
1180 C(I,J) = 0
1190 FOR K = 1 TO N3
1200 C(I,J) = C(I,J) + A(I,K) * B(K,J)
1210 NEXT K
1220 NEXT J
1230 NEXT I
1240 RETURN
```

A must be an N1 by N3 matrix; B must be an N3 by N2 matrix. The resultant matrix C will be an N1 and N2 matrix. A, B, and C must be dimensioned accordingly.

7/Arithmetic Functions

Model III BASIC offers a wide variety of intrinsic ("built-in") functions for performing arithmetic and special operations. The special-operation functions are described in the next chapter.

All the common math functions described in this chapter return single-precision values ***accurate to six decimal places.*** *ABS, FIX and INT return values whose precision depends on the precision of the argument.*

The conversion functions (CINT, CDBL, etc.) return values whose precision depends on the particular function. Trig functions use or return radians, not degrees. A radian-degree conversion is given for each of the functions.

For all the functions, the argument must be enclosed in parentheses. The argument may be either a numeric variable, expression or constant.

Functions described in this chapter:

ABS	COS	INT	SGN
ATN	CSNG	LOG	SIN
CDBL	EXP	RANDOM	SQR
CINT	FIX	RND	TAN

ABS *(x)*

Returns the absolute value of the argument. ABS(X) = X for X greater than or equal to zero, and ABS(X) = – X for X less than zero.

```
100 IF ABS(X)<1E-6 PRINT "TOO SMALL"
```

ATN *(x)*

Returns the arctangent (in radians) of the argument; that is, ATN(X) returns "the angle whose tangent is X". To get arctangent in degrees, multiply ATN(X) by 57.29578.

```
100 Y=ATN(B/C)
```

CDBL *(x)*

Returns a double-precision representation of the argument. The value returned will contain 17 digits, but only the digits contained in the argument will be significant.

CDBL may be useful when you want to force an operation to be done in double-precision, even though the operands are single precision or even integers. For example CDBL (I%)/J% will return a fraction with 17 digits of precision.

```
100 FOR I% = 1 TO 25 : PRINT 1/CDBL(I%), : NEXT
```

CINT *(x)*

Returns the largest integer not greater than the argument. For example, CINT (1.5) returns 1; CINT(− 1.5) returns − 2. For the CINT function, the argument must be in the range − 32768 to + 32767. The result is stored internally as a two-byte integer.

CINT might be used to speed up an operation involving single or double-precision operands without losing the precision of the operands (assuming you're only interested in an integer result).

```
100 K% = CINT(X#) + CINT(Y#)
```

COS *(x)*

Returns the cosine of the argument (argument must be in radians). To obtain the cosine of X when X is in degrees, use COS(X*.01745329).

```
100 Y = COS(X + 3.3)
```

CSNG *(x)*

Returns a single-precision representation of the argument. When the argument is a double-precision value, it is returned as six significant digits with "4/5 rounding" in the least significant digit. So CSNG(.6666666666666667) is returned as .666667; CSNG(.3333333333333333) is returned as .333333.

```
100 PRINT CSNG (A# + B#)
```

EXP *(x)*

Returns the "natural exponential" of X, that is e^X. This is the inverse of the LOG function, so X = EXP(LOG(X)).

```
100 PRINT EXP(-X)
```

FIX *(x)*

Returns a truncated representation of the argument. All digits to the right of the decimal point are simply chopped off, so the resultant value is an integer. For non-negative X, FIX(X) = INT(X). For negative values of X, FIX(X) = INT(X) + 1. For example, FIX(2.2) returns 2, and FIX(− 2.2) returns − 2.

```
100 Y = ABS(A - FIX(A))
```

This statement gives Y the value of the fractional portion of A.

INT*(x)*

Returns an integer representation of the argument, using the largest whole number that is not greater than the argument. Argument is **not** limited to the range − 32768 to + 32767. The result is stored internally as a single-precision whole number. INT(2.5) returns 2; INT(− 2.5) returns − 3; and INT(1000101.23) returns 1000101.

```
100 Z = INT(A*100 + .5)/100
```

Gives Z the value of A rounded to two decimal places (for non-negative A).

LOG*(x)*

Returns the natural logarithm of the argument, that is, $\log_e$ *(argument)*. This is the inverse of the EXP function, so X = LOG (EXP (X)). To find the logarithm of a number to another base *b*, use the formula $LOG_b(X) = LOG_e(x)/LOG_e(b)$. For example, LOG(32767)/LOG(2) returns the logarithm to base 2 of 32767.

```
100 PRINT LOG(3.3*X)
```

RANDOM

RANDOM is actually a complete statement rather than a function. It reseeds the random number generator. If a program uses the RND function, you may want to put RANDOM at the beginning of the program. This will ensure that you get an unpredictable sequence of pseudo-random numbers each time you turn on the Computer, load the program, and run it.

```
10 RANDOM
20 PRINT RND(0);
30 GOTO 20          'DO LINE 10 JUST ONCE
```

RND*(x)*

Generates a pseudo-random number using the current pseudo-random "seed number" (generated internally and not accessible to user). RND may be used to produce random numbers between 0 and 1, or random integers greater than 0, depending on the argument.

RND(0) returns a single-precision value between 0 and 1. RND(*integer*) returns an integer between 1 and *integer* inclusive (*integer* must be positive and less than 32768). For example, RND(55) returns a pseudo-random integer greater than zero and less than 56. RND(55.5) returns a number in the same range, because RND uses the INTeger value of the argument.

```
100  X=RND(2):ON X GOTO 200,300
```

SGN*(x)*

The "sign" function: returns −1 for X negative, 0 for X zero, and +1 for X positive.

```
100  ON SGN(X)+2 GOTO 200,300,400
```

SIN*(x)*

Returns the sine of the argument (argument must be in radians). To obtain the sine of X when X is in degrees, use SIN(X*.01745329).

```
100   PRINT SIN(A*B – B)
```

SQR*(x)*

Returns the square root of the argument. SQR(X) is the same as X[(1/2), only faster.

```
100 Y = SQR(X[ 2 – H[ 2)
```

TAN*(x)*

Returns the tangent of the argument (argument must be in radians). To obtain the tangent of X when X is in degrees, use TAN(X*.01745329).

```
100   Z = TAN(2*A)
```

NOTE: A great many other functions may be created using the above functions. See Appendix E, ''Derived Functions''.

8/Special Features

Model III BASIC offers some unusual functions and operations that deserve special highlighting. Some may seem highly specialized; as you learn more about programming and begin to experiment with machine-language routines, they will take on more significance. Other functions in the chapter are of obvious benefit and will be used often (for example, the graphics functions).

Functions, statements and operators described in this chapter:

Graphics:	Error-Routine Functions:	Other Functions and Statements:
SET	ERL	INP
RESET	ERR	MEM
CLS		OUT
POINT		PEEK
		POKE
		POS
		USR
		VARPTR

SET*(x,y)*

Turns on the graphics block at the location specified by the coordinates *x* and *y*. For graphics purposes, the Display is divided up into a 128 (horizontal) by 48 (vertical) grid. The *x*-coordinates are numbered from left to right, 0 to 127. The *y*-coordinates are numbered from top to bottom, 0 to 47. Therefore the point at (0,0) is in the extreme upper left of the Display, while the point at (127,47) is in the extreme lower right corner. See the Video Display Worksheet in the Appendix.

The arguments *x* and *y* may be numeric constants, variables or expressions. They need not be integer values, because SET(*x,y*) uses the INTeger portion of *x* and *y*. SET (*x,y*) is valid for:

$0<=x<128$
$0<=y<48$

Examples:

```
100 SET (RND(128) – 1,RND(48) – 1)
```

Lights up a random point on the Display.

```
100 INPUT X,Y: SET (X,Y)
```

RUN to see where the blocks are.

RESET *(x,y)*

Turns off a graphics block at the location specified by the coordinates *x* and *y*. This function has the same limits and parameters as SET*(x,y)*.

```
200 RESET (X,3)
```

CLS

"Clear-Screen" — turns off all the graphics blocks on the Display and moves the cursor to the upper left corner. This wipes out alphanumeric characters as well as graphics blocks. CLS is very useful whenever you want to present an attractive Display output.

```
10 CLS
20 SET(RND(128)-1, RND(48)-1)
30 GOTO 20
```

POINT*(x,y)*

Tests whether the specified graphics block is "on" or "off". If the block is "on" (that is, if it has been SET), then POINT returns a binary True (– 1 in Model III BASIC). If the block is "off", POINT returns a binary False (0 in Model III BASIC). Typically, the POINT test is put inside an IF-THEN statement.

```
100 SET (50, 28) : IF POINT (50,28) THEN PRINT "ON" ELSE PRINT "OFF"
```

This line will always print the message, "ON", because POINT(50,28) will return a binary True, so that execution proceeds to the THEN clause. If the test failed, POINT would return a binary False, causing execution to jump to the ELSE statement.

ERL

Returns the line number in which an error has occurred. This function is primarily used inside an error-handling routine accessed by an ON ERROR GOTO statement. If no error has occurred when ERL is called, line number 0 is returned. However, if an error has occurred since power-up, ERL returns the line number in which the error occurred. If error occurred in direct mode, 65535 is returned (largest number representable in two bytes).

Example Program using ERL

```
10 CLEAR 10
20 ON ERROR GOTO 1000
30 INPUT "ENTER YOUR MESSAGE"; M$
40 INPUT "NOW ENTER A NUMBER"; N
50 Z = 1/N
60 PRINT "INPUT VALUES OKAY--TRY AGAIN TO CAUSE AN ERROR"
70 GOTO 30
1000 IF ERL=30 AND (ERR/2 + 1 = 14) THEN 1040
1010 IF ERL=40 AND (ERR/2 + 1 = 6) THEN 1050
1020 IF ERL=50 AND (ERR/2 + 1 = 11) THEN 1060
1030 ON ERROR GOTO 0: RESUME
1040 PRINT "MESSAGE TOO LONG--10 LETTERS MAXIMUM": RESUME
1050 PRINT "NUMBER TOO LARGE": RESUME
1060 PRINT "DIVISION BY ZERO IN LINE 50--ENTER NON-ZERO NUMBER"
1070 RESUME 40
```

RUN the program. Try entering a long message; try entering zero when the program asks for a number. Note that ERL is used in line 1000 to determine where the error occurred so that appropriate action may be taken.

ERR/2 + 1

Similar to ERL, except ERR returns a value **related** to the **code** of the error rather than the line in which the error occurred. It is commonly used inside an error handling routine accessed by an ON ERROR GOTO statement. See Appendix B, "Error Codes."

ERR/2 + 1 = true error code
(true error code – 1)*2 = ERR

Sample Program

See ERL.

INP *(port)*

Returns a byte-value from the specified port. There are 256 ports, numbered 0-255. For example

```
100     PRINT INP(50)
```

inputs a byte from port 50 and prints the decimal value of the byte.

You do **not** need to access the Z-80 ports to make full use of the TRS-80.

MEM

Returns the number of unused and unprotected bytes in memory. This function may be used in the Immediate Mode to see how much space a resident program takes up; or it may be used inside the program to avert OM (Out of Memory) errors by allocating less string space, DIMensioning smaller array sizes, etc. MEM requires no argument.

Example:

```
100     IF MEM < 80 THEN 900
```

Enter the command PRINT MEM (in the Immediate Mode) to find out the amount of memory not being used to store programs, variables, strings, stack, or reserved for object-files.

OUT *port, value*

Outputs a byte value to the specified port. OUT is not a function but a statement complete in itself. It requires two arguments separated by a comma (no parenthesis): the port destination and the byte value to be sent.
port and *value* are in the range 0 to 255.

PEEK*(address)*

Returns the value stored at the specified byte address (in decimal form). To use this function, you'll need to refer to two sections of the Appendix: the Memory Map (so you'll know where to PEEK) and the Table of Function ASCII and Graphics Codes (so you'll know what the values represent).

If you're using PEEK to examine object files, you'll also need a microprocessor instruction set manual (one is included with the TRS-80 Editor/Assembler Instruction Manual).

PEEK is valuable for linking machine language routines with Model III BASIC programs. The machine language routine can store information in a certain memory location, and PEEK may be used inside your BASIC program to retrieve the information. For example,

```
A = PEEK (17999)
```

returns the value stored at location 17999 and assigns that value to the variable A.

Peek may also be used to retrieve information stored with a POKE statement. Using PEEK and POKE allows you to set up very compact, byte-oriented storage systems. Refer to the Memory Map in the Appendix to determine the appropriate locations for this type of storage. See **POKE, USR.**

POKE *address, value*

Loads a value into a specified memory location. POKE is not a function but a statement complete in itself. It requires two arguments: a byte address (in decimal form) and a value. The value must be between 0 and 255 inclusive. Refer to the Memory Map in the Appendix to see which addresses you'd like to POKE.

To POKE (or PEEK) an address **above** 32767, use the following formula: $-1 *$ (65536-desired address) = POKE OR PEEK address. For example, to POKE into address 32769, use POKE -32767, *value*.

Since the Video Display is memory-mapped, you can output to the Display directly by POKEing ASCII data into Video RAM. Video RAM is from 15360 to 16383.
Example:

```
10 CLS
20 FOR M = 15360 TO 16383
30 POKE M,191
40 NEXT M
50 GOTO 50
```

RUN the program to see how fast the screen is ''painted'' white.

Since POKE can be used to store information anywhere in memory, it is very important when we do our graphics to stay in the range for display locations. If we POKE outside this range, we may store the byte in a critical place. We could be POKEing into our program, or even in worse places like the stack. Indiscriminate POKEing can be disastrous. You might have to reset or power off and start over again. Unless you know where you are POKEing — don't.

See **PEEK, USR, SET,** and **CHR$** for background material. Also see the Owners Section for examples on special uses of POKE.

POS(*x*)

Returns a number from 0 to 63 indicating the current cursor position on the Display. Requires a ''dummy argument'' (any numeric expression).

```
100   PRINT TAB(40); POS(0)
```

prints 40 at position 40. (Note that a blank is inserted before the ''4'' to accommodate the sign; therefore the ''4'' is actually at position 41.) The ''0'' in ''POS(0)'' is the dummy argument.

USR *(x)*

This function lets you call a machine-language subroutine and then continue execution of your BASIC program.

"Machine language" is the low-level language used internally by your Computer. It consists of Z-80 microprocessor instructions. Machine-language subroutines are useful for special applications (things you can't do in BASIC) and simply because they can do things very fast (like white-out the Display).

Writing such routines requires familiarity with assembly-language programming and with the Z-80 instruction set. For more information on this subject, see the Radio Shack book, *TRS-80 Assembly-Language Programming,* by William Barden, Jr., and the instruction manual for Radio Shack's EDITOR-ASSEMBLER (26-2002).

Getting the USR routine into memory

1. You should first reserve the area in high memory where the routine will be located. This is done immediately after power-up by answering the MEMORY SIZE? question with the address **preceding the start address of your USR routine.** For example, if your routine starts at 32700, then type 32699 in response to MEMORY SIZE?.
2. Then load the routine into memory.
   A. If it is stored on tape in the SYSTEM format (created with EDITOR-ASSEMBLER), you must load it via the SYSTEM command, as described in Chapter 2. After the tape has loaded press (BREAK) to return to the BASIC immediate mode.
   B. If it is a short routine, you may simply want to POKE it into high memory.

Telling BASIC where the USR routine starts

Before you can make the USR call, you have to tell BASIC the entry address to the routine. Simply POKE the two-byte address into memory locations 16526-16527: least significant byte (LSB) into 16526, most significant byte (MSB) into 16527.

For example, if the entry point is at 32700:
32700 decimal = 7FBC hexadecimal
LSB = BC hexadecimal = 188 decimal
MSB = 7F hexadecimal = 127 decimal
So use the statements:

```
POKE 16526, 188
POKE 16527, 127
```

to tell BASIC that the USR routine entry is at 32700.

Making the USR call

At the point in your BASIC program where you want to call the subroutine, insert a statement like

X = USR(N)

where N can be an expression and must have a value between − 32768 and + 32767 inclusive. This argument, N, can be used to pass a value to your routine (see below) or you can simply consider it a dummy argument and not use it at all.

When BASIC encounters your X = USR(N) statement, it will branch to the address stored at 16526-16527. At the point in your USR routine where you want to **return** to the BASIC program, insert a simple RET instruction — unless you want to return a value to BASIC, in which case, see below.

Passing an argument to the USR routine

If you want to pass the USR(N) argument to your routine, then include the following CALL instruction at the **beginning** of your USR routine.:

```
CALL 0A7FH
```

This loads the argument N into the HL register pair as a two-byte signed integer.

Returning an argument from the USR routine

To return an integer value to the USR(N) function, load the value (a two-byte signed integer) into HL and place the following jump instruction at the end of your routine:

```
JP    0A9AH
```

Control will pass back to your program, and the integer in HL will replace USR(N). For example, if the call was

```
X = USR(N)
```

Then X will be given the value in HL.

USR routines are automatically allocated up to 8 stack levels or 16 bytes (a high and low memory byte for each stack level). If you need more stack space, you can save the BASIC stack pointer and set up your own stack. See **SYSTEM, PEEK,** and **POKE.** Also see the Technical Information Chapter in the Owners Section.

VARPTR *(variable name)*

Returns an address-value which will help you locate where the variable name and its value are stored in memory. If the variable you specify has not been assigned a value, an FC error will occur when this function is called.

If VARPTR*(integer variable)* returns address K:
Address K contains the least significant byte (LSB) of 2-byte *integer*.
Address K + 1 contains the most significant byte (MSB) of *integer*.

You can display these bytes (two's complement decimal representation) by executing a PRINT PEEK (K) and a PRINT PEEK (K + 1).

If VARPTR*(single precision variable)* returns address K:
(K)* = LSB of value
(K + 1) = Next most significant byte (Next MSB)
(K + 2) = MSB with hidden (implied) leading one. Most significant bit is the sign of the number
(K + 3) = exponent of value excess 128 (128 is added to the exponent).

If VARPTR*(double precision variable)* returns K:
(K) = LSB of value
(K + 1) = Next MSB
(K + ...) = Next MSB
(K + 6) = MSB with hidden (implied) leading one. Most significant bit is the sign of the number.
(K + 7) = exponent of value excess 128 (128 is added to the exponent).

For single and double precision values, the number is stored in normalized exponential form, so that a decimal is assumed before the MSB. 128 is added to the exponent. Furthermore, the high bit of MSB is used as a sign bit. It is set to 0 if the number is positive or to 1 if the number is negative. See examples below.

You can display these bytes by executing the appropriate PRINT PEEK*(x)* where *x* = the address you want displayed. Remember, the result will be the decimal representation of byte, with bit 7 (MSB) used as a sign bit. The number will be in normalized exponential form with the decimal assumed before the MSB. 128 is added to the exponent,

If VARPTR*(string variable)* returns K:
K = length of string
(K + 1) = LSB of string value starting address
(K + 2) = MSB of string value starting address
*** (K) signifies "contents of address K"**

The address will probably be in high RAM where string storage space has been set aside. But, if your string variable is a constant (a string literal), then it will point to the area of memory where the program line with the constant is stored, in the program buffer area. Thus, program statements like A$ = "HELLO" do not use string storage space.

For all of the above variables, addresses (K – 1) and (K – 2) will store the TRS-80 Character Code for the variable name. Address (K – 3) will contain a descriptor code that tells the Computer what the variable type is. Integer is 02; single precision is 04; double precision is 08; and string is 03.

VARPTR*(array variable)* will return the address for the first byte of that element in the array. The element will consist of 2 bytes if it is an integer array; 3 bytes if it is a string array; 4 bytes if it is a single precision array; and 8 bytes if it is a double precision array.

The first element in the array is preceded by:

1. A sequence of two bytes per dimension, each two-byte pair indicating the "depth" of each respective dimension.
2. A single byte indicating the total number of dimensions in the array.
3. A two-byte pair indicating the total number of elements in the array.
4. A two-byte pair containing the ASCII-coded array name.
5. A one-byte type-descriptor (02 = Integer, 03 = String, 04 = Single-Precision, 08 = Double-Precision).

Item (1) immediately precedes the first element, Item (2) precedes Item (1), and so on.

The elements of the array are stored sequentially with the first dimension-subscripts varying "fastest", then the second, etc.

Examples:

A! = 2 will be stored as follows
2 = 10 Binary, represented as .1E2 = $.1 \times 2^2$
So exponent of A is 128 + 2 = 130 (called excess 128)
MSB of A is 10000000;
however, the high bit is changed to zero since the value is positive (called hidden or implied leading one).
So A! is stored as

Exponent (K + 3)	MSB (K + 2)	Next MSB (K + 1)	LSB (K)
130	0	0	0

A! = – .5 will be stored as

Exponent (K + 3)	MSB (K + 2)	Next MSB (K + 1)	LSB (K)
128	128	0	0

A! = 7 will be stored as

Exponent (K + 3)	MSB (K + 2)	Next MSB (K + 1)	LSB (K)
131	96	0	0

A! = –7:

Exponent (K + 3)	MSB (K + 2)	Next MSB (K + 1)	LSB (K)
131	224	0	0

Zero is simply stored as a zero-exponent. The other bytes are insignificant.

9/Editing

You have probably found it is very time consuming to retype long program lines, simply because of a typo, or maybe just to make a small change.

Model III editing features eliminate much of this extra work. In fact, it's so easy to alter program lines, you'll probably want to experiment with multi-statement lines, complex expressions, etc.

Commands, subcommands, and special function keys described in this chapter:

EDIT	(L)	*n*(D)
(ENTER)	(X)	*n*(C)
n(SPACEBAR)	(I)	*n*(S)c
n (←)	(A)	*n*(K)c
(SHIFT)(↑)	(E)	
	(Q)	
	(H)	

EDIT *line number*

This command puts you in the Edit Mode. You must specify which line you wish to edit, in one of two ways:

EDIT *line-number*	Lets you edit the specified line. If line number is not in use, an FC error occurs
or	
EDIT.	Lets you edit the current program line — last line entered or altered or in which an error has occurred.

For example, type in and (ENTER) the following line:

```
100 FOR I = 1 TO 10 STEP .5 : PRINT I, I [2, I [3 : NEXT
```

This line will be used in exercising all the Edit subcommands described below.

Now type EDIT 100 and hit (ENTER). The Computer will display:

```
100■
```

You are now in the Edit Mode and may begin editing line 100.

NOTE: EDITing a program line automatically clears all variable values and eliminates pending FOR/NEXT and GOSUB operations. If BASIC encounters a syntax error during program execution, it will automatically put you in the EDIT mode. Before EDITing the line, you may want to examine current variable values. In this case, you must type Q as your first EDIT command. This will return you to the command mode, where you may examine variable values. Any other EDIT command (typing E, pressing ENTER, etc.) will clear out all variables.

(ENTER) key

Hitting (ENTER) while in the Edit Mode causes the Computer to record all the changes you've made (if any) in the current line, and returns you to the Command Mode.

n(SPACEBAR)

In the Edit Mode, hitting the Space-bar moves the cursor over one space to the right and displays any character stored in the preceding position. For example, using line 100 entered above, put the Computer in the Edit Mode so the Display shows:

```
100■
```

Now hit the Space-Bar. The cursor will move over one space, and the first character of the program line will be displayed. If this character was a blank, then a blank will be displayed. Hit the Space-Bar until you reach the first non-blank character:

```
100 F■
```

is displayed. To move over more than one space at a time, hit the desired number of spaces first, and then hit the space-bar. For example, type 5 and hit Space-bar, and the display will show something like this (may vary depending on how many blanks you inserted in the line):

```
100 FOR I = ■
```

Now type 8 and hit the Space-bar. The cursor will move over 8 spaces to the right, and 8 more characters will be displayed.

n ←

Moves the cursor to the left by *n* spaces. If no number *n* is specified, the cursor moves back one space. When the cursor moves to the left, all characters in its "path" are erased from the display, but **they are not deleted from the program line.** Using this in conjunction with D or K or C can give misleading Video Displays of your program lines. So, be careful using it! For example, assuming you've used *n*Space-Bar so that the Display shows:

```
100 FOR I = 1 TO 10■
```

type 8 and hit the ← key. The display will show something like this:

```
100 FOR I = ■
```

(will vary depending on number of blanks in your line 100)

SHIFT ↑

Hitting SHIFT and ↑ keys together effects an escape from any of the Insert subcommands: X, I and H. After escaping from an Insert subcommand, you'll still be in the Edit Mode, and the cursor will remain in its current position. (Hitting ENTER is another way to exit these Insert subcommands).

L (List Line)

When the Computer is in the Edit Mode, and is not currently executing one of the subcommands below, hitting L causes the remainder of the program line to be displayed. The cursor drops down to the next line of the Display, reprints the current line number, and moves to the first position of the line. For example, when the Display shows

```
100■
```

hit L (without hitting ENTER key) and line 100 will be displayed:

```
100 FOR I = 1 TO 10 STEP .5 : PRINT I, I [2, I [3 : NEXT
100■
```

This lets you look at the line in its current form while you're doing the editing.

X (Extend Line)

Causes the rest of the current line to be displayed, moves cursor to end of line, and puts Computer in the Insert subcommand mode so you can add material to the end of the line. For example, using line 100, when the Display shows

```
100■
```

hit X (without hitting (ENTER))and the entire line will be displayed; notice that the cursor now follows the last character on the line:

```
100 FOR I = 1 TO 10 STEP .5 : PRINT I, I [2, I [3 : NEXT■
```

We can now add another statement to the line, or delete material from the line by using the (X) key. For example, type :PRINT"DONE" at the end of the line. Now hit (ENTER). If you now type LIST 100, the Display should show something like this:

```
100 FOR 1 = 1 TO 10 STEP .5: PRINT 1,1 [ 2, 1 [ 3:NEXT:PRINT "DONE"
```

I (Insert)

Allows you to insert material beginning at the current cursor position on the line. (Hitting (I) will actually delete material from the line in this mode.) For example, type and (ENTER) the EDIT 100 command, then use the Space Bar to move over to the decimal point in line 100. The Display will show:

```
100 FOR I = 1 TO 10 STEP .■
```

Suppose you want to change the increment from .5 to .25. Hit the I key (don't hit (ENTER)) and the Computer will now let you insert material at the current position. Now hit 2 so the Display shows:

```
100 FOR I = 1 TO 10 STEP .2■
```

You've made the necessary change, so hit (SHIFT) (↑) to escape from the Insert Subcommand. Now hit L key to display remainder of line and move cursor back to the beginning of the line:

```
100 FOR I = 1 TO 10 STEP .25 : PRINT I, I [ 2, I [3 : NEXT : PRINT "DONE"
100■
```

You can also exit the Insert subcommand and save all changes by hitting (ENTER) . This will return you to Command mode.

A (Cancel and Start Again)

Moves the cursor back to the beginning of the program line and cancels editing changes already made. For example, if you have added, deleted, or changed something in a line, and you wish to go back to the beginning of the line and cancel the changes already made: first hit SHIFT (↑) (to escape from any subcommand you may be executing); then hit A. (The cursor will drop down to the next line, display the line number and move to the first program character.)

E (Exit)

Causes Computer to end editing and save all changes made. You must be in Edit Mode, not executing any subcommand, when you hit E to end editing.

Q (Quit)

Tells Computer to end editing and cancel all changes made in the current editing session. If you've decided not to change the line, type Q to cancel changes and leave Edit Mode.

H (Hack)

Tells Computer to delete remainder of line and lets you insert material at the current cursor position. Hitting (H) will actually delete a character from the line in this mode. For example, using line 100 listed above, enter the Edit Mode and space over to the last statement, PRINT"DONE". Suppose you wish to delete this statement and insert an END statement. Display will show:

```
100 FOR I = 1 TO 10 STEP .25 : PRINT I, I [2, I [3 : NEXT : ■
```

Now type H and then type END. Hit (ENTER) key. List the line:

```
100 FOR I = 1 TO 10 STEP .25 : PRINT I, I [2, I [3 : NEXT : END
```

should be displayed.

*n*D (Delete)

Tells Computer to delete the specified number *n* characters to the right of the cursor. The deleted characters will be enclosed in exclamation marks to show you which characters were affected. For example, using line 100, space over to the PRINT command statement:

```
100 FOR I = 1 TO 10 STEP .25 :■
```

Now type 19D. This tells the Computer to delete 19 characters to the right of the cursor. The display should show something like this:

```
100 FOR I = 1 TO 10 STEP .25 : !PRINT I, I [2, I [3 :!■
```

When you list the complete line, you'll see that the PRINT statement has been deleted.

*n*C (Change)

Tells the Computer to let you change the specified number of characters beginning at the current cursor position. If you type C without a preceding number, the Computer assumes you want to change one character. When you have entered *n* number of characters, the Computer returns you to the Edit Mode (so you're not in the *n*C Subcommand). For example, using line 100, suppose you want to change the final value of the FOR-NEXT loop, from "10" to "15". In the Edit Mode, space over to just before the "0" in "10".

```
100 FOR I = 1 TO 1■
```

Now type C. Computer will assume you want to change just one character. Type 5, then hit L. When you list the line, you'll see that the change has been made.

```
100 FOR 1 = 1 TO 15 STEP .25 : NEXT : END
```

would be the current line if you've followed the editing sequence in this chapter.

The ⊖ does not work as a backspace under the C command in Edit mode. Instead, it replaces the character you want to change with a backspace. So it should not be used. If you make a mistake while typing in a change, Edit the line again to correct it, instead of using ⊖.

*n*S*c* (Search)

Tells the Computer to search for the *n*th occurrence of the character *c*, and move the cursor to that position. If you don't specify a value for *n*, the Computer will search for the first occurrence of the specified character. If character *c* is not found, cursor goes to the end of the line. Note: The Computer only searches through characters to the right of the cursor.

For example, using the current form of line 100, type EDIT 100 ((ENTER)) and then hit 2S: . This tells the Computer to search for the second occurrence of the colon character. Display should show:

```
100 FOR I = 1 TO 15 STEP .25 : NEXT ■
```

You may now execute one of the subcommands beginning at the current cursor position. For example, suppose you want to add the counter variable after the NEXT statement. Type I to enter the Insert subcommand, then type the variable name, I. That's all you want to insert, so hit SHIFT (↑) to escape from the Insert subcommand. The next time you list the line, it should appear as:

```
100 FOR I = 1 TO 15 STEP .25 : NEXT I : END
```

*n*K*c* (Kill)

Tells the Computer to delete all characters up to the *n*th occurrence of character *c*, and move the cursor to that position. For example, using the current version of line 100, suppose we want to delete the entire line up to the END statement. Type EDIT 100 ((ENTER)), and then type 2K:. This tells the Computer to delete all characters up to the 2nd occurrence of the colon. Display should show:

```
100 !FOR I = 1 TO 15 STEP .25 : NEXT I!■
```

The second colon still needs to be deleted, so type D. The Display will now show:

```
100 !FOR I = 1 TO 15 STEP .25 : NEXT I!!:!■
```

Now hit (ENTER) and type LIST 100 ((ENTER)).

Line 100 should look something like this:

```
100 END
```

Section 3: Appendices

A / Model III Summary

Special Characters and Abbreviations

Command Mode	Function
(ENTER)	Return carriage and interpret command
←	Cursor backspace and delete last character typed
(SHIFT)←	Cursor to beginning of line; erase line
↓	Linefeed
:	Statement delimiter; use between statements on same logical line
→	Move cursor to next tab stop. Tab stops are at positions 0, 8, 16, 24, 32, 48, and 56.
(SHIFT)→	Convert display to 32 characters per line
(CLEAR)	Clear Display and convert to 64 characters per line

Execute Mode	Function
(SHIFT)@	Pause in execution; freeze display during LIST
(BREAK)	Stop execution
(ENTER)	Interpret data entered from Keyboard with INPUT statement

Abbreviations	Function
?	Use in place of PRINT.
'	Use in place of :REM
.	"current line"; use in place of line number with LIST, EDIT, etc.

To output a control character, press (SHIFT) then (↓); while holding down both keys, press the key for which a control character is desired. For example, to key a control —Z press:

(SHIFT)(↓)(Z)

Type Declaration Characters

Character	Type	Examples	Section 2 Page
$	String	A$, ZZ$	1/13
%	Integer	A1%, SUM%	1/13
!	Single-Precision	B!, NI!	1/12
#	Double-Precision	A#, 1/3#	1/12
D	Double-Precision (exponential notation)	1.23456789D-12	1/12
E	Single-Precision (exponential notation)	1.23456E+30	1/12

Arithmetic Operators

		Section 2 Page
+	add	1/19
−	subtract	1/19
*	multiply	1/19
/	divide	1/19
[	exponentiate (e.g., 2[3 = 8) Press ⓐ to generate "[".	1/19

String Operator

			Section 2 Page
+	concatenate (string together)	"2" + "2" = "22"	1/22

Relational Operators

Symbol	in numeric expressions	in string expressions	Section 2 Page
<	is less than	precedes	1/23
>	is greater than	follows	1/23
=	is equal to	equals	1/23
< = or = <	is less than or equal to	precedes or equals	1/23
> = or = >	is greater than or equal to	follows or equals	1/23
<> or ><	does not equal	does not equal	1/23

Order of Operations	Section 2 Page
[or ↑ (Exponentiation) Press ↑ to enter this character.	1/26
− (Negation)	1/26
*,/	1/26
+, −	1/26
Relational operators	1/26
NOT	1/26
AND	1/26
OR	1/26
Precedence order is from left to right for operators on the same level	1/26

Commands

Command/Function	Examples	Section 2 Page
AUTO *mm, nn* Turn on automatic line numbering beginning with *mm*, using increment of *nn*.	AUTO AUTO 10 AUTO 5,5 AUTO .,10	2/1
CLEAR Set numeric variables to zero, strings to null.	CLEAR	
CLEAR *n* Same as CLEAR but also sets aside *n* bytes for strings.	CLEAR 500 CLEAR MEM/4	2/2
CLOAD Load a BASIC program from tape	CLOAD"A"	2/2
CLOAD? Verifies BASIC program on tape to one in memory	CLOAD?"A"	2/3
CONT Continue after BREAK or STOP in execution.	CONT	2/3

CSAVE Save a BASIC program on tape	CSAVE"A"	2/3
DELETE *mm-nn* Delete program line from line *mm* to line *nn*.	DELETE 100 DELETE 10-50 DELETE.	2/4
EDIT *mm* Enter Edit Mode for line *mm*. See Edit Mode Sub-commands below.	EDIT 100 EDIT.	2/4
LIST *mm-nn* List all program lines from *mm* to *nn*.	LIST LIST 30-60 LIST 30- LIST-90 LIST .	2/4
LLIST *mm-nn* Lists all program lines from *mm* to *nn* on the line printer.	LLIST LLIST 30-60	2/5
NEW Delete entire program and reset all variables, pointers etc.	NEW	2/5
RUN *mm* Execute program beginning at lowest numbered line or *mm* if specified.	RUN RUN 55	2/6
SYSTEM Enter Monitor Mode for loading of machine-language file from cassette.	SYSTEM	2/6
TROFF Turn off Trace	TROFF	2/7
TRON Turn on Trace	TRON	2/7

Edit Mode Subcommands and Functions

Sub-Command	Function	Section 2 Page
(ENTER)	End editing and return to Command Mode.	9/2
(SHIFT) (↑)	Escape from X, I, and H subcommands and remain in Edit Mode.	9/3
n(SPACEBAR)	Move cursor *n* spaces to right.	9/2
n(←)	Move cursor *n* spaces to left.	9/3
(L)	List remainder of program line and return to beginning of line.	9/3
(X)	List remainder of program line, move cursor to end of line, and start Insert subcommand.	9/4
(I)	Insert the following sequence of characters at current cursor position; use Escape to exit this subcommand.	9/4
(A)	Cancel changes and return cursor to beginning of line	9/5
(E)	End editing, save all changes and return to Command Mode.	9/5
(Q)	End editing, cancel all changes made and return to Command Mode.	9/5
(H)	Delete remainder of line and insert following sequence of characters; use Escape to exit this subcommand.	9/5
n(D)	Delete specified number of characters *n* beginning at current cursor position.	9/6
n(C)	Change (or replace) the specified number of characters *n* using the next *n* characters entered.	9/6
n(S)*c*	Move cursor to *n*th occurrence of character *c*, counting from current cursor position.	9/7
n(K)*c*	Delete all characters from current cursor position up to *n*th occurrence of character *c*, counting from current cursor position.	9/7

Input/Output Statements

Statement/Function	Examples	Section 2 Page
PRINT *exp**		
Output to Display the value of *exp*. *Exp* may be a numeric or string expression or constant, or a list of such items.	PRINT A$ PRINT X+3 PRINT "D=" D	3/1
Comma serves as a PRINT modifier. Causes cursor to advance to next print zone.	PRINT 1,2,3,4 PRINT "1", "2" PRINT 1,,2	
Semi-colon serves as a PRINT modifier. Inserts a space after a numeric item in PRINT list. Inserts no space after a string item. At end of PRINT list, suppresses the automatic carriage return.	PRINT X;" = ANSWER" PRINT X;Y;Z PRINT "ANSWER IS";	
PRINT@*n*		
PRINT modifier; begin PRINTing at specified display position *n*.	PRINT @ 540, "CENTER" PRINT @ N+3,X*3	3/2
PRINT TAB *n*		
Print modifier: moves cursor to specified Display position *n* (expression).	PRINT TAB(N) N	3/3
PRINT USING *string;exp*		
PRINT format specifier; output *exp* in form specified by *string* field (see below).	PRINT USING A$;X PRINT USING "#.#";Y+Z	3/4
INPUT *''message'';variable*		
Print message (if any) and await input from Keyboard.	INPUT"ENTER NAME";A$ INPUT"VALUE";X INPUT"ENTER NUMBERS" ;X,Y INPUT A,B,C,D$	3/8
LPRINT		
Output to line printer.	LPRINT A$	3/12
PRINT # − 1		
Output to Cassette.	PRINT # − 1,A,B,C,D$	3/12

**exp* may be a string of numeric constant or variable, or a list of such items.

INPUT # – 1 Input from Cassette.	INPUT # – 1,A,B,C,D$	3/13
DATA *item list* Hold data for access by READ statement.	DATA 22,33,11,1.2345 DATA "HALL", "SMITH", "DOE"	3/10
READ *variable list* Assign value(s) to the specified variable(s), starting with current DATA element.	READ A,A1,A2,A3 READ A$,B$,C$,D	3/10
RESTORE Reset DATA pointer to first item in first DATA statement.	RESTORE	3/11

Field Specifiers for PRINT USING statements

Numeric Character	Function	Example	Section 2 Page
#	Numeric field (one digit digit per #).	###	3/4
.	Decimal point position.	##.###	3/4
+	Print leading or trailing signs (plus for positive numbers, minus for negative numbers).	+#.### #.###+	3/5
–	Print trailing sign only if value printed is negative.	###.##–	3/5
**	Fill leading blanks with asterisk.	**###.##	3/4
$$	Place dollar sign immediately to left of leading digit.	$$####.##	3/4
**$	Dollar sign to left of leading digit and fill leading blanks with asterisks.	**$####.##	3/4
[[[[or ⓐⓐⓐⓐ	Exponential format, with one significant digit to left of decimal. Press ⓐ to input this character.	#.##[[[[	3/4

,	Prints out number with commas, as in 1,356,000	#,######	**3**/4
!	Single character.	!	**3**/5
%spaces%	String with length equal to 2 plus number of spaces between % symbols.	%%	**3**/5

Program Statements

Statement/Function	Examples	Section 2 Page
(Type Definition)		
DEFDBL *letter list or range* Define as double-precision all variables beginning with specified letter, letters or range of letters.	DEFDBL J DEFDBL X,Y,A DEFDBL A-E,J	**4**/3
DEFINT *letter list or range* Define as integer all variables beginning with specified letter, letters or range of letters.	DEFINT A DEFINT C,E,G DEFINT A-K	**4**/2
DEFSNG *letter list or range* Define as single-precision all variables beginning with specified letter, letters or range of letters	DEFSNG L DEFSNG A-L, Z DEFSNG P,R,A-K	**4**/2
DEFSTR *letter list or range* Define as string all variables beginning with the specified letter, letters, or range of letters.	DEFSTR A-J	**4**/3
(Assignment and Allocation)		
CLEAR *n* Set aside specified number of bytes *n* for string storage. Clears value and type of all variables.	CLEAR 750 CLEAR MEM/10 CLEAR 0	**4**/4

DIM *array(dim#1,...,dim#k)* Allocate storage for *k*-dimensional *array* with the specified size per dimension: dim #1, dim#2,..., etc. DIM may be followed by a list of arrays separated by commas.	DIM A(2,3) DIM A1(15), A2(15) DIM B(X + 2),C(J,K) DIM T(3,3,5)	4/4
LET *variable* = *expression* Assign value of *expression* to *variable*. LET is optional in LEVEL II BASIC.	LET A$ = "CHARLIE" LET B1 = C1 LET A% = 1#	4/5
(Sequence of Execution)		
END End execution, return to Command Mode.	99 END	4/5
STOP Stop execution, print Break message with current line number. User may continue with CONT.	100 STOP	4/6
GOTO *line-number* Branch to specified *line-number*.	GOTO 100	4/6
GOSUB *line-number* Branch to sub-routine beginning at *line-number*.	GOSUB 3000	4/7
RETURN Branch to statement following last-executed GOSUB.	RETURN	4/7
ON *exp* GOTO *line#1,...,line#k* Evaluate expression; if INT (exp) equals one of the numbers 1 through *k*, branch to the appropriate line number. Otherwise go to next statement.	ON K + 1 GOTO 100,200,300	4/8
ON *exp* GOSUB *line#1,...,line#k* Same as ON...GOTO except branch is sub-routine beginning at *line#1, line#2,...,* or *line#k*, depending on *exp*.	ON J GOSUB 330,700	4/9

Statement/Functions	Examples	Section 2 Page
FOR *var* = *exp* TO *exp* STEP *exp* Open a FOR-NEXT loop. STEP is optional; if not used, increment of one is used.	FOR I = 1 TO 50 STEP 1.5 FOR M% = J% TO K% – 1	**4**/9
NEXT *variable* Close FOR-NEXT loop. *Variable* may be omitted. To close nested loops, a variable list may be used. See Chapter 4.	NEXT NEXT I NEXT I,J,K	**4**/9
ERROR *(code)* Simulate the error specified by *code* (See Error Code Table).	ERROR(14)	**4**/12
ON ERROR GOTO *line-number* If an error occurs in subsequent program lines, branch to error routine beginning at *line-number*.	ON ERROR GOTO 999	**4**/12
RESUME *n* Return from error routine to line specified by *n*. If *n* is zero or not specified, return to statement containing error. If *n* is "NEXT", return to statement following error-statement.	RESUME RESUME 0 RESUME 100 RESUME NEXT	**4**/3
RANDOM Reseeds random number generator.	RANDOM	**7**/4
REM REMark indicator; ignore rest of line.	REM A IS ALTITUDE	**4**/14

(Tests – Conditional Statements)

IF *exp-1* THEN *statement-1*
ELSE *statement-2*

Tests *exp-1:* If True, execute *statement-1* then jump to next program line (unless statement-1 was a GOTO). If *exp-1* is False, jump directly to ELSE statement and execute subsequent statements.	IF A = 0 THEN PRINT "ZERO" ELSE PRINT "NOT ZERO"	**4**/14-**4**/15

(Graphics Statements)

CLS

Clear Video Display	CLS	**8**/2

RESET*(x,y)*

Turn off the graphics block with horizontal coordinate *x* and vertical coordinate *y*, 0< = X<128 and 0< = Y<48	RESET (8 + B,11)	**8**/2

SET *(x,y)*

Turn on the graphics block specified by coordinates *x* and *y*. Same argument limits as RESET	SET(A*2,B + C)	**8**/1

(Special Statements)

POKE *location, value*

Load *value* into memory *location* (both arguments in decimal form) 0< = *value*< = 255.	POKE 15635,34 POKE 17770,A + N	**8**/5

OUT *port, value*

Send *value* to *port* (both arguments between 0 and 255 inclusive)	OUT 255,10 OUT 55,A	**8**/5

String Functions*

Function	Operation	Examples	Section 2 Page
ASC*(string)*	Returns ASCII code of first character in string argument.	ASC(B$) ASC("H")	5/2
CHR$*(code exp)*	Returns a one-character string defined by *code*. If *code* specifies a control function, that function is activated.	CHR$(34) CHR$(1)	5/2
FRE*(string)*	Returns amount of memory available for string storage. Argument is a dummy variable.	FRE(A$)	5/3
INKEY$	Strobes Keyboard and returns a one-character string corresponding to key pressed during strobe (null string if no key is pressed).	INKEY$	5/4
LEFT$*(string,n)*	Returns first *n* characters of *string*.	LEFT$(A$,1) LEFT$(L1$ + C$,8) LEFT$(A$,M + L)	5/5
LEN*(string)*	Returns length of *string* (zero for null string).	LEN(A$ + B$) LEN("HOURS")	5/5
MID$*(string,p,n,)*	Returns substring of *string* with length *n* and starting at position *p* in *string*.	MID$(M$,5,2) MID$(M$ + B$,P,L − 1)	5/6
RIGHT$*(string,n)*	Returns last *n* characters of *string*.	RIGHT$(NA$,7) RIGHT$(AB$,M2)	5/6
STR$*(numeric exp)*	Returns a string representation of the evaluated argument.	STR$(1.2345) STR$(A + B*2)	5/6
STRING$*(n,char)*	Returns a sequence of *n char* symbols using first character of *char*.	STRING$(30, ".") STRING$(25, "A") STRING$(5,C$)	5/7
TIME$	Returns date and time.	TIME$	5/8
VAL*(string)*	Returns a numeric value corresponding to a numeric-valued string.	VAL("1" + A$ + "." + C$) VAL(A$ + B$) VAL(G1$)	5/8

**string* may be a string variable, expression, or constant.

Arithmetic Functions*

Function	Operation (unless noted otherwise, $-1.7E+38 <= exp <= 1.7E+38$)	Examples	Section 2 Page
ABS*(exp)*	Returns absolute value.	ABS(L*.7) ABS(SIN(X))	7/1
ATN*(exp)*	Returns arctangent in radians.	ATN(2.7) ATN(A*3)	7/1
CDBL*(exp)*	Returns double-precision representation of *exp*.	CDBL(A) CDBL(A + 1/3#)	7/2
CINT*(exp)*	Returns largest integer not greater than *exp*. Limits: $-32768 <= exp < +32768$.	CINT(A# + B)	7/2
COS*(exp)*	Returns the cosine of *exp*; assumes *exp is in radians*.	COS(2*A) COS(A/57.29578)	7/2
CSNG*(exp)*	Returns single-precision representation, with 5/4 rounding in least significant decimal when *exp* is double-precision.	CSNG(A#) CSNG(.33*B#)	7/2
EXP*(exp)*	Returns the natural exponential, $e^{exp} = \text{EXP}(exp)$.	EXP(34.5) EXP(A*B*C − 1)	7/3
FIX*(exp)*	Returns the integer equivalent to truncated *exp* (fractional part of *exp* is chopped off).	FIX(A − B)	7/3
INT*(exp)*	Returns largest integer not greater than *exp*.	INT(A + B*C)	7/3
LOG*(exp)*	Returns natural logarithm (base e) of *exp*. Limits: *exp* must be positive.	LOG(12.33) LOG(A B + B)	7/3
RND(0)	Returns a pseudo-random number between 0.000001 and 0.999999 inclusive.	RND(0)	7/4
RND*(exp)*	Returns a pseudo-random number between 1 and INT*(exp)* inclusive. Limits: $1 <= exp < 32768$.	RND(40) RND(A + B)	7/4
SGN*(exp)*	Returns − 1 for negative *exp*; 0 for zero *exp*; + 1 for positive *exp*.	SGN(A*B + 3) SGN(COS(X))	7/4

**exp* is any numeric-valued expression or constant.

Function	Operation	Examples	Section 2 Page
SIN*(exp)*	Returns the sine of *exp*; assumes *exp* is in radians.	SIN(A/B) SIN(90/57.29578)	7/5
SQR*(exp)*	Returns square root of *exp*. Limits: *exp* must be non-negative.	SQR(A*A – B*B)	7/5
TAN*(exp)*	Returns the tangent of *exp;* assumes *exp* is in radians.	TAN(X) TAN(X*.01745329)	7/5

Special Functions

Function	Operation and Limits	Examples	Section 2 Page
ERL	Returns line number of current error.	ERL	8/3
ERR	Returns a value related to current error code (if error has occurred). ERR = (error code – 1)*2. Also: (ERR/2) + 1 = error code.	ERR/2 + 1	8/3
INP*(port)*	Inputs and returns the current value from the specified *port*. Both argument and result are in the range 0 to 255 inclusive.	INP(55)	8/4
MEM	Returns total unused and unprotected bytes in memory. Does not include unused string storage space.	MEM	8/4
PEEK*(location)*	Returns value stored in the specified memory byte. *location* must be a valid memory address in decimal form (see Memory Map in Appendix D).	PEEK(15370)	8/4
POINT*(x,y)*	Checks the graphics block specified by horizontal coordinate *x* and vertical coordinate *y*. If block is "on", returns a True (–1); if block is "off", returns a False (0). Limits: 0<=*x*<128;0<=*y*<48.		8/2
POS(0)	Returns a number indicating the current cursor position. The argument "0" is a dummy variable.	POS(0)	8/4
USR*(n)*	Branches to machine language subroutine. See Chapter 8.	USR(0)	8/7
VARPTR*(var)*	Returns the address where the specified variable's name, value and pointer are stored, *var* must be a valid variable name.	VARPTR(A$) VARPTR(N1)	8/9

Model III BASIC Reserved Words*

@
ABS
AND
ASC
ATN
AUTO
CDBL
CHR$
CINT
CLEAR
CLOCK
CLOSE
CLS
CMD
CONT
COS
CSNG
CVD
CVI
CVS
DATA
DEFDBL
DEFFN
DEFINT
DEFSNG
DEFUSR
DEFSTR
DELETE
DIM
EDIT
ELSE
END
EOF
ERL
ERR
ERROR
EXP
FIELD
FIX
FN
FOR
FORMAT
FRE
FREE
GET
GOSUB
GOTO
IF
INKEY$
INP
INPUT
INSTR
INT
KILL
LEFT$
LET
LSET
LEN
LINE
LIST
LLIST
LPRINT
LOAD
LOC
LOF
LOG
MEM
MERGE
MID$
MKD$
MKI$
MKS$
NAME
NEW
NEXT
NOT
ON
OPEN
OR
OUT
PEEK
POINT
POKE
POS
POSN
PRINT
PUT
RANDOM
READ
REM
RENAME
RESET
RESTORE
RESUME
RETURN
RIGHT$
RND
RSET
RUN
SAVE
SET
SGN
SIN
SQR
STEP
STOP
STRING$
STR$
SYSTEM
TAB
TAN
THEN
TIME$
TO
TROFF
TRON
USING
USR
VAL
VARPTR
VERIFY

*Some of these words have no function in Model III BASIC; they are reserved for use in Disk BASIC. None of these words can be used inside a variable name. You'll get a syntax error if you try to use these words as variables.

Program Limits and Memory Overhead

Ranges

Integers −32768 to +32767 inclusive
Single Precision −1.701411E±38 to +1.701411E±38 inclusive
Double Precision −1.701411834544556D±38 to +1.701411834544556D±38 inclusive

String Range: Up to 255 characters

Line Numbers Allowed: 0 to 65529 inclusive

Program Line Length: Up to 255 characters (input 240, edit to 255)

Memory Overhead

Program lines require 5 bytes minimum, as follows:
- Line Number — 2 bytes
- Line Pointer — 2 bytes
- Carriage Return — 1 byte

In addition, each reserved word, operator, variable name, special character and constant character requires one byte.

Dynamic (RUN-Time) Memory Allocation

Integer variables: 5 bytes each
(2 for value, 3 for variable name)

Single-precision variables: 7 bytes each
(4 for value, 3 for variable name)

Double-precision variables: 11 bytes each
(8 for value, 3 for variable name)

String variables: 6 bytes minimum
(3 for variable name, 3 for stack and variable pointers, 1 for each character)

Array variables: 12 bytes minimum
(3 for variable name, 2 for total size, 1 for number of dimensions, 2 for size of each dimension, and 2, 3, 4 or 8 [depending on array type]
for each element in the array)

Each active FOR-NEXT loop requires 16 bytes.

Each active (non-returned) GOSUB requires 6 bytes.

Each level of parentheses requires 4 bytes plus 12 bytes for each temporary value.

General Formula for Computing Memory Requirements of Arrays

The array G (N1, N2, ..., Nk) requires the following amount of memory:

$$14 + (k*2) + T*\{(N1+1)*(N2+1)*...*(Nk+1)\}$$

where k is the number of dimensions in the array, and the value of T depends on the array type:

Type	T =
Integer	2
Single-Precision	4
Double-Precision	8
String*	3

*In computing the actual memory requirements of string arrays, you must add the text length of each element in the array. When the array is first dimensioned, all elements have length 0. The string text will be stored in the string space (reserved by the CLEAR n statement).

Accuracy

Single-precision calculations involving +, −, *, and / are accurate to six significant digits; double-precision calculations involving the same operations are accurate to 16 significant digits.

The exponentiation operator ↑ (displayed as "[") is single-precision.

The trigonometric and logarithmic functions are single-precision; other functions have a precision depending on the input argument and on the function. For example, CDBL returns a double-precision value; ABS returns a value with the same precision as the input argument.

When converting from single- to double-precision, use the following technique to avoid introduction of incorrect values in the extra digits of precision:

double-precision variable = VAL (STR$ (*Single-precision variable*))

B / Error Codes

CODE	ABBREVIATION	ERROR
1	NF	NEXT without FOR
2	SN	Syntax error
3	RG	Return without GOSUB
4	OD	Out of data
5	FC	Illegal function call
6	OV	Overflow
7	OM	Out of memory
8	UL	Undefined line
9	BS	Subscript out of range
10	DD	Redimensioned array
11	/0	Division by zero
12	ID	Illegal direct
13	TM	Type mismatch
14	OS	Out of string space
15	LS	String too long
16	ST	String formula too complex
17	CN	Can't continue
18	NR	NO RESUME
19	RW	RESUME without error
20	UE	Unprintable error
21	MO	Missing operand
22	FD	Bad file data
23	L3	Disk BASIC only

Explanation of Error Messages

NF NEXT without FOR: NEXT is used without a matching FOR statement. This error may also occur if NEXT *variable* statements are reversed in a nested loop.

SN Syntax Error: This usually is the result of incorrect punctuation, open parenthesis, an illegal character or a mis-spelled command.

RG RETURN without GOSUB: A RETURN statement was encountered before a matching GOSUB was executed.

OD Out of Data. A READ or INPUT # statement was executed with insufficient data available. DATA statement may have been left out or all data may have been read from tape or DATA.

FC Illegal Function Call: An attempt was made to execute an operation using an illegal parameter. Examples: square root of a negative argument, negative matrix dimension, negative or zero LOG arguments, etc. Or USR call without first POKEing the entry point.

OV Overflow: The magnitude of the number input or derived is too large for the Computer to handle. NOTE: There is no underflow error. Numbers smaller than ±1.701411E−38 single precision or ±1.701411834544556E−38 double precision are rounded to 0. See /0 below.

OM Out of Memory: All available memory has been used or reserved. This may occur with very large matrix dimensions, nested branches such as GOTO, GOSUB, and FOR-NEXT Loops.

UL Undefined Line: An attempt was made to refer or branch to a non-existent line.

BS Subscript out of Range: An attempt was made to assign a matrix element with a subscript beyond the DIMensioned range.

DD Redimensioned Array: An attempt was made to DIMension a matrix which had previously been dimensioned by DIM or by default statements. It is a good idea to put all dimension statements at the beginning of a program.

/0 Division by Zero: An attempt was made to use a value of zero in the denominator. NOTE: If you can't find an obvious division by zero check for division by numbers smaller than allowable ranges. See OV above and RANGES page A/17.

ID Illegal Direct: The use of INPUT as a direct command.

TM Type Mismatch: An attempt was made to assign a non-string variable to a string or vice-versa.

OS Out of String Space: The amount of string space allocated was exceeded.

LS String Too Long: A string variable was assigned a string value which exceeded 255 characters in length.

ST String Formula Too Complex: A string operation was too complex to handle. Break up the operation into shorter steps.

CN Can't Continue: A CONT was issued at a point where no continuable program exists, e.g., after program was ENDed or EDITed.

NR No RESUME: End of program reached in error-trapping mode.

RW RESUME without ERROR: A RESUME was encountered before ON ERROR GOTO was executed.

UE Unprintable Error: An attempt was made to generate an error using an ERROR statement with an invalid code.

MO Missing Operand: An operation was attempted without providing one of the required operands.

FD Bad File Data: Data input from an external source (i.e., tape) was not correct or was in improper sequence, etc.

L3 DISK BASIC only: An attempt was made to use a statement, function or command which is available only with the Disk System.

C / TRS-80 Model III Character Codes

Text is represented in the Computer by codes. For example, the letter "A" is represented by the code 65. **Control functions and graphics** are also represented by codes. The character codes range from zero through 255.

Codes **zero through 31** usually represent certain **control functions.** For example, code 13 represents a carriage return or "end of line". However, in the Model III, these same codes also represent 32 special display characters. For this application, they must be loaded (POKEd) into video RAM, not PRINTed.

Codes **32 through 127** represent the **text characters** — all those letters, numbers and other characters that are commonly used to represent textual information. The Model III text characters conform to the American National Standard Code for Information Interchange.

Codes **128 through 191**, when output to the video display, represent 64 **graphics** characters.

Codes **192 through 255**, when output to the video display, represent either **space compression codes** or **special characters,** as determined by software.

Many of the codes may be input from the keyboard; all of them may be stored in a string and output to any device. For example, to output a code 31 to the video display, use a statement like this:

```
PRINT CHR$(31)
```

For further details, see Using the Video Display in Section One of this manual.

Note: In the following table, *vidram* refers to Video RAM, i.e., addresses from 15360 to 16383.

In the following table, we summarize the keyboard and video display control characters.

Code Dec.	Code Hex.	Keyboard	Video Display PRINT CHR$ *(code)*	POKE *vidram, code**
1	00		No effect	See **Special Characters** 0 through 31 later in this Appendix.
1	01	BREAK SHIFT ↓ A	No effect	
2	02	SHIFT ↓ B	No effect	
3	03	SHIFT ↓ C	No effect	
4	04	SHIFT ↓ D	No effect	
5	05	SHIFT ↓ E	No effect	
6	06	SHIFT ↓ F	No effect	
7	07	SHIFT ↓ G	No effect	
8	08	← SHIFT ↓ H	Backspace and erase	
9	09	→ SHIFT ↓ I	Tab (0, 8, 16, 24, . . .)	
10	0A	↓ SHIFT ↓ J	Move cursor to start of next line and erase line	
11	0B	SHIFT ↓ K	No effect	
12	0C	SHIFT ↓ L	No effect	
13	0D	ENTER SHIFT ↓ M	Move cursor to start of next line and erase line	
14	0E	SHIFT ↓ N	Cursor on	
15	0F	SHIFT ↓ O	Cursor off	
16	10	SHIFT ↓ P	No effect	
17	11	SHIFT ↓ Q	No effect	
18	12	SHIFT ↓ R	No effect	
19	13	SHIFT ↓ S	No effect	
20	14	SHIFT ↓ T	No effect	
21	15	SHIFT ↓ U	Swap space compression/ special characters	
22	16	SHIFT ↓ V	Swap special/alternate characters	
23	17	SHIFT ↓ W	Double-size characters	
24	18	SHIFT ← SHIFT ↓ X	Backspace without erasing	
25	19	SHIFT ↓ Y	Advance cursor	
26	1A	SHIFT ↓ Z	Move cursor down	
27	1B	SHIFT ↑	Move cursor up	
28	1C	SHIFT ↓ ,	Move cursor to upper left corner	
29	1D	SHIFT ↓ 9	Erase line and start over	
30	1E	SHIFT ↓ .	Erase to end of line	
31	1F	CLEAR SHIFT ↓ /	Erase to end of display	

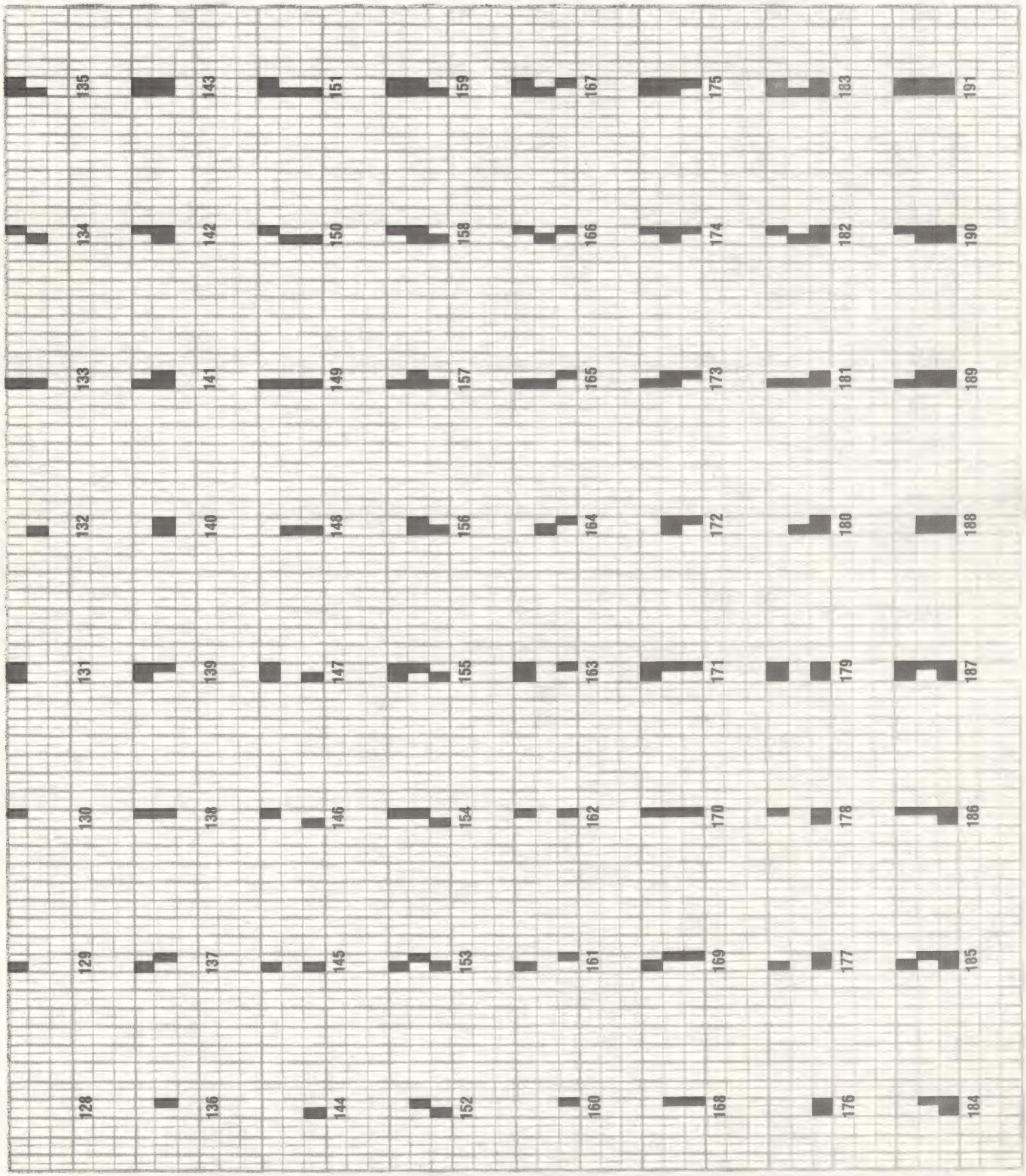
*See Special Characters 0 through 31 later in this Appendix.

Code		Key-board	Video Display	
Dec.	Hex.		PRINT CHR$ *(code)*	POKE *vidram, code*
32	20	SPACEBAR	ø	ø
33	21	!	!	!
34	22	"	"	"
35	23	#	#	#
36	24	$	$	$
37	25	%	%	%
38	26	&	&	&
39	27	'	'	'
40	28	(	(	(
41	29	)	)	)
42	2A	*	*	*
43	2B	+	+	+
44	2C	,	,	,
45	2D	–	–	–
46	2E	.	.	.
47	2F	/	/	/
48	30	0	0	0
49	31	1	1	1
50	32	2	2	2
51	33	3	3	3
52	34	4	4	4
53	35	5	5	5
54	36	6	6	6
55	37	7	7	7
56	38	8	8	8
57	39	9	9	9
58	3A	:	:	:
59	3B	;	;	;
60	3C	<	<	<
61	3D	=	=	=
62	3E	>	>	>
63	3F	?	?	?
64	40	@	@	@
65	41	A	A	A
66	42	B	B	B
67	43	C	C	C
68	44	D	D	D

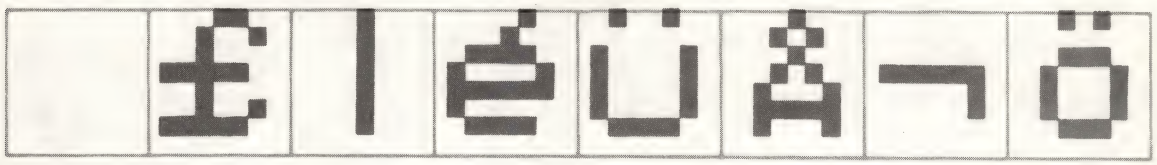
Code		Key-board	Video Display	
Dec.	Hex.		PRINT CHR$ *(code)*	POKE *vidram, code*
69	45	E	E	E
70	46	F	F	F
71	47	G	G	G
72	48	H	H	H
73	49	I	I	I
74	4A	J	J	J
75	4B	K	K	K
76	4C	L	L	L
77	4D	M	M	M
78	4E	N	N	N
79	4F	O	O	O
80	50	P	P	P
81	51	Q	Q	Q
82	52	R	R	R
83	53	S	S	S
84	54	T	T	T
85	55	U	U	U
86	56	V	V	V
87	57	W	W	W
88	58	X	X	X
89	59	Y	Y	Y
90	5A	Z	Z	Z
91	5B	↑	[	[
92	5C		\	\
93	5D		]	]
94	5E		^	^
95	5F		—	—
96	60	SHIFT @	`	`
97	61	A	a	a
98	62	B	b	b
99	63	C	c	c
100	64	D	d	d
101	65	E	e	e
102	66	F	f	f
103	67	G	g	g
104	68	H	h	h
105	69	I	i	i

Code		Key-board	Video Display	
Dec.	Hex.		PRINT CHR$ *(code)*	POKE *vidram, code*
106	6A	J	j	j
107	6B	K	k	k
108	6C	L	l	l
109	6D	M	m	m
110	6E	N	n	n
111	6F	O	o	o
112	70	P	p	p
113	71	Q	q	q
114	72	R	r	r
115	73	S	s	s
116	74	T	t	t
117	75	U	u	u
118	76	V	v	v
119	77	W	w	w
120	78	X	x	x
121	79	Y	y	y
122	7A	Z	z	z
123	7B		{	{
124	7C		¦	¦
125	7D		}	}
126	7E		~	~
127	7F		±	±
128	80	Codes 128-191 output graphics characters. See the graphic display table in this Appendix.		
192 : :	C0	Codes 192-255 output either space compression codes or special characters when used with PRINT CHR$ *(code)*.		
255	FF	They always output special characters when used with POKE *vidram, code*. See the special character table in this Appendix.		

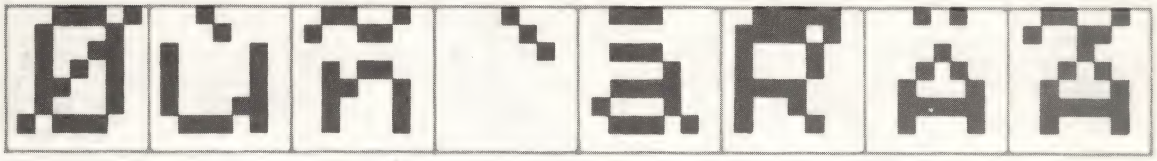
Graphics Characters (Codes 128-191)



Special Characters (0-31, 192-255)

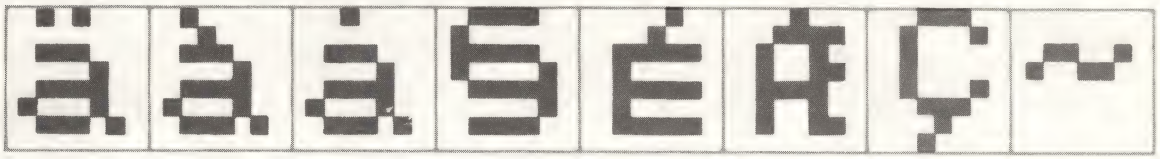


0 1 2 3 4 5 6 7

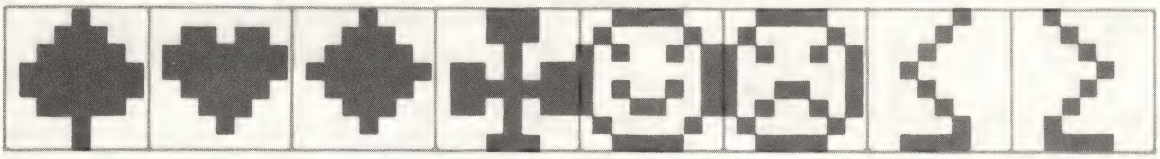


8 9 10 11 12 13 14 15

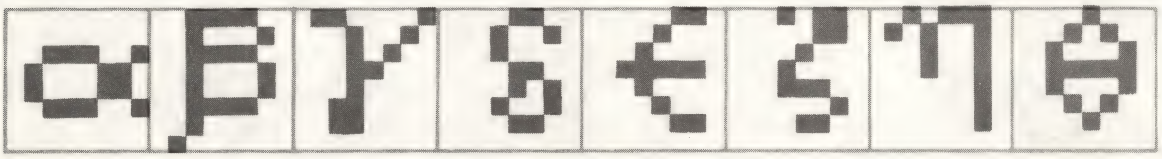
16 17 18 19 20 21 22 23



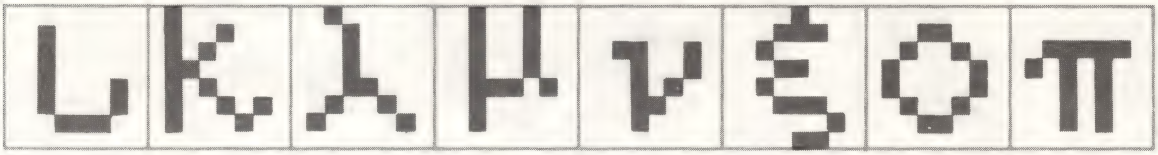
24 25 26 27 28 29 30 31



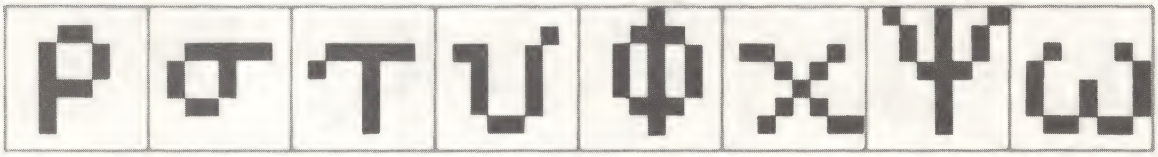
192 193 194 195 196 197 198 199



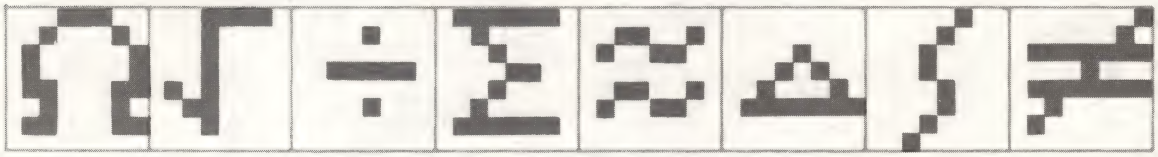
200 201 202 203 204 205 206 207



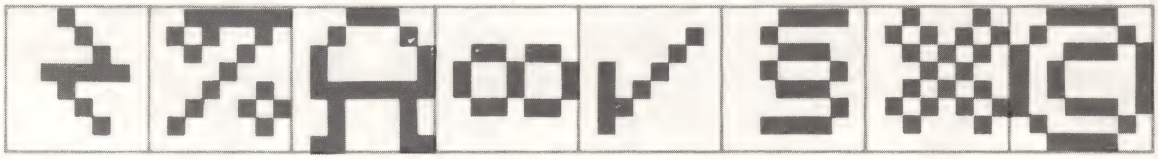
208 209 210 211 212 213 214 215



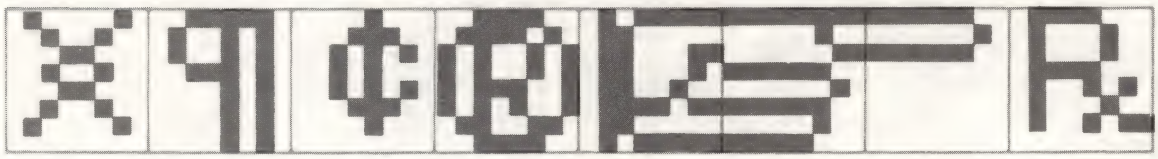
216 217 218 219 220 221 222 223



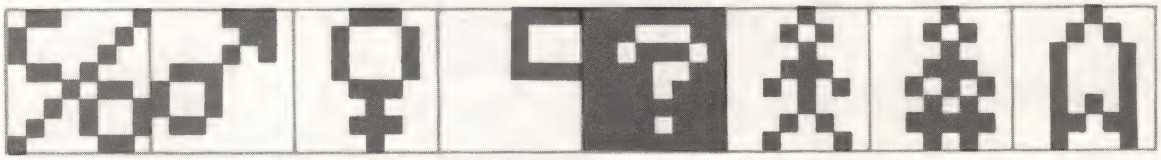
224 225 226 227 228 229 230 231



232 233 234 235 236 237 238 239



240 241 242 243 244 245 246 247

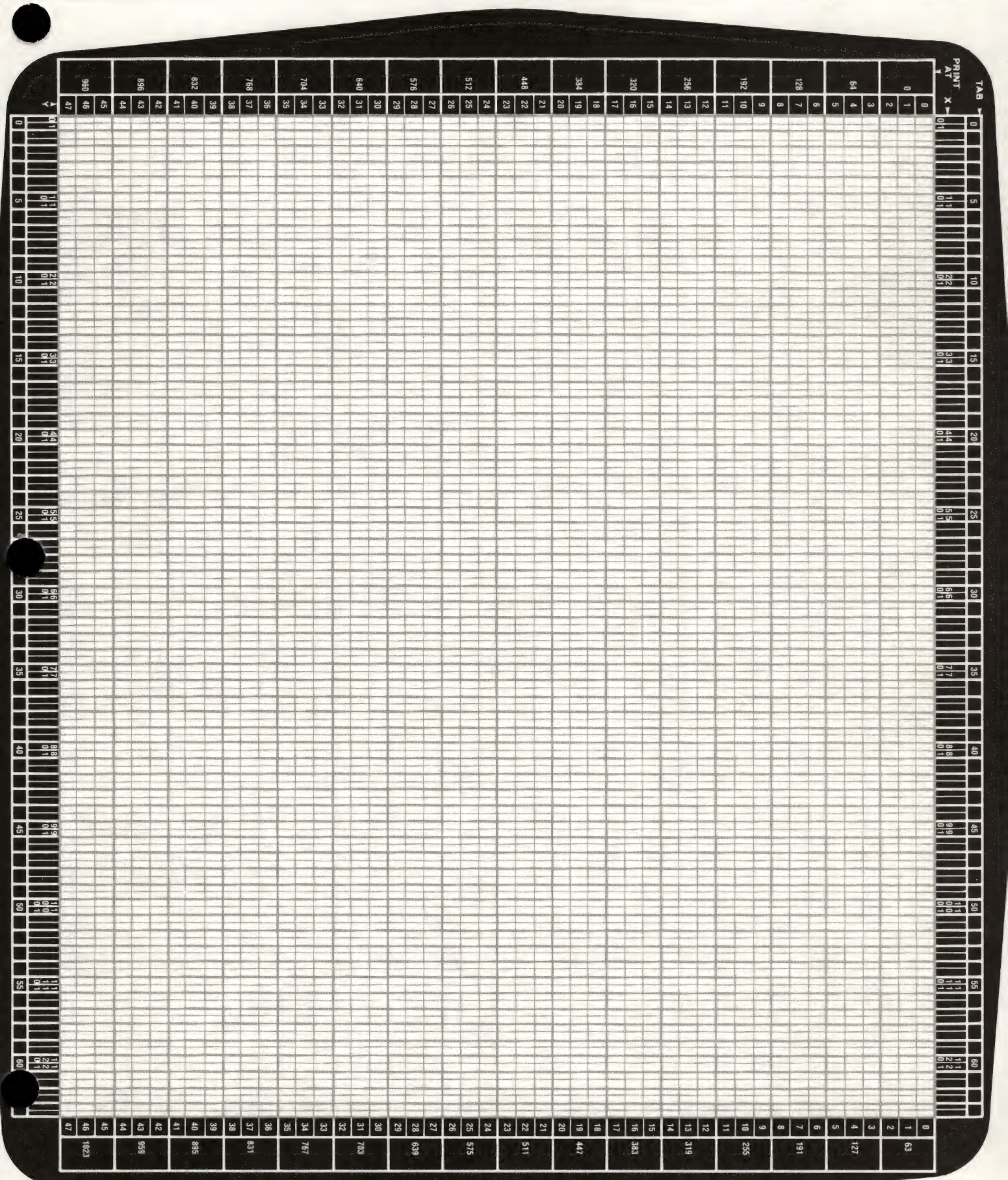


248 249 250 251 252 253 254 255

TRS-80 Video Display Worksheet

Radio Shack®

TITLE _____ PROGRAMMER _____ PAGE ____ OF ____



D / Internal Codes for BASIC Keywords

The following are the internal codes that the Computer uses to store BASIC keywords. If you PEEK at the program buffer area (starting at address 17129 in decimal) you will find your program stored in the following codes.

Dec. Code	BASIC Keyword	Dec. Code	BASIC Keyword
129	FOR	167	LOAD
130	RESET	168	MERGE
131	SET	169	NAME
132	CLS	170	KILL
133	CMD	171	LSET
134	RANDOM	172	RSET
135	NEXT	173	SAVE
136	DATA	174	SYSTEM
137	INPUT	175	LPRINT
138	DIM	176	DEF
139	READ	177	POKE
140	LET	178	PRINT
141	GOTO	179	CONT
142	RUN	180	LIST
143	IF	181	LLIST
144	RESTORE	182	DELETE
145	GOSUB	183	AUTO
146	RETURN	184	CLEAR
147	REM	185	CLOAD
148	STOP	186	CSAVE
149	ELSE	187	NEW
150	TRON	188	TAB
151	TROFF	189	TO
152	DEFSTR	190	FN
153	DEFINT	191	USING
154	DEFSNG	192	VARPTR
155	DEFDBL	193	USR
156	LINE	194	ERL
157	EDIT	195	ERR
158	ERROR	196	STRING$
159	RESUME	197	INSTR
160	OUT	198	POINT
161	ON	199	TIME$
162	OPEN	200	MEM
163	FIELD	201	INKEY$
164	GET	202	THEN
165	PUT	203	NOT
166	CLOSE	204	STEP

Dec. Code	BASIC Keyword	Dec. Code	BASIC Keyword
205	+	231	CVS
206	−	232	CVD
207	*	233	EOF
208	/	234	LOC
209		235	LOF
210	AND	236	MKI$
211	OR	237	MKS$
212	>	238	MKD$
213	=	239	CINT
214	<	240	CSNG
215	SGN	241	CDBL
216	INT	242	FIX
217	ABS	243	LEN
218	FRE	244	STR$
219	INP	245	VAL
220	POS	246	ASC
221	SQR	247	CHR$
222	RND	248	LEFT$
223	LOG	249	RIGHT$
224	EXP	250	MID$
225	COS		
226	SIN		
227	TAN		
228	ATN		
229	PEEK		
230	CVI		

E / Derived Functions

Function	Function Expressed in Terms of Model III BASIC Functions. X is in radians.
SECANT	SEC(X) = 1/COS(X)
COSECANT	CSC(X) = 1/SIN(X)
COTANGENT	COT(X) = 1/TAN(X)
INVERSE SINE	ARCSIN(X) = ATN(X/SQR(−X*X+1))
INVERSE COSINE	ARCCOS(X) = −ATN(X/SQR(−X*X+1))+1.5708
INVERSE SECANT	ARCSEC(X) = ATN(SQR(X*X−1))+(SGN(X)−1)*1.5708
INVERSE COSECANT	ARCCSC(X) = ATN(1/SQR(X*X−1))+(SGN(X)−1)*1.5708
INVERSE COTANGENT	ARCCOT(X) = −ATN(X)+1.5708
HYPERBOLIC SINE	SINH(X) = (EXP(X)−EXP(−X))/2
HYPOBOLIC COSINE	COSH(X) = (EXP(X)+EXP(−X))/2
HYPERBOLIC TANGENT	TANH(X) = −EXP(−X)/(EXP(X)+EXP(−X))*2+1
HYPERBOLIC SECANT	SECH(X) = 2/(EXP(X)+EXP(−X))
HYPERBOLIC COSECANT	CSCH(X) = 2/(EXP(X)−EXP(−X))
HYPERBOLIC COTANGENT	COTH(X) = EXP(−X)/(EXP(X)−EXP(−X))*2+1
INVERSE HYPERBOLIC SINE	ARGSINH(X) = LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	ARGCOSH(X) = LOG(X+SQR(X*X−1))
INVERSE HYPERBOLIC TANGENT	ARGTANH(X) = LOG((1+X)/(1−X))/2
INVERSE HYPERBOLIC SECANT	ARGSECH(X) = LOG((SQR(−X*X+1)+1)/X)
INVERSE HYPERBOLIC COSECANT	ARGCSCH(X) = LOG((SGN(X)*SQR(X*X+1)+1)/X)
INVERSE HYPERBOLIC COTANGENT	ARGCOTH(X) = LOG((X+1)/(X−1))/2

Valid Input Ranges

Inverse Sine	−1 < X < 1
Inverse Cosine	−1 < X < 1
Inverse Secant	X < − or X>1
Inverse Cosecant	X < −1 or X > 1
Inverse Hyper. Cosine	X > 1
Inverse Hyper. Tangent	X*X < 1
Inverse Hyper. Secant	0 < X < 1
Inverse Hyper. Cosecant	X <> 0
Inverse Hyper. Cotangent	X*X > 1

Certain special values are mathematically undefined, but our functions may provide invalid values:

TAN and SEC of 90 and 270 degrees
COT and CSC of 0 and 180 degrees

For example, TAN(1.5708) returns a value but TAN(90* .01745329) returns a DIVISION BY ZERO error. 90* .01745329 = 1.5708

Other values which are not available from these functions are:

```
ARSCIN  (-1) = - PI / 2
ARCSIN  (  1) =   PI / 2
ARCCOS (-1) =   PI
ARCCOS (  1) =   0
ARCSEC (-1) = - PI
ARCSEC (  1) =   0
ARCCSC (-1) = - PI / 2
ARCCSC (  1) =   PI / 2
```

Please note that the above information may not be exhaustive.

F / Base Conversions

DEC.	HEX.	BINARY
0	00	00000000
1	01	00000001
2	02	00000010
3	03	00000011
4	04	00000100
5	05	00000101
6	06	00000110
7	07	00000111
8	08	00001000
9	09	00001001
10	0A	00001010
11	0B	00001011
12	0C	00001100
13	0D	00001101
14	0E	00001110
15	0F	00001111
16	10	00010000
17	11	00010001
18	12	00010010
19	13	00010011
20	14	00010100
21	15	00010101
22	16	00010110
23	17	00010111
24	18	00011000
25	19	00011001
26	1A	00011010
27	1B	00011011
28	1C	00011100
29	1D	00011101
30	1E	00011110
31	1F	00011111
32	20	00100000
33	21	00100001
34	22	00100010
35	23	00100011
36	24	00100100
37	25	00100101
38	26	00100110
39	27	00100111

DEC.	HEX.	BINARY
40	28	00101000
41	29	00101001
42	2A	00101010
43	2B	00101011
44	2C	00101100
45	2D	00101101
46	2E	00101110
47	2F	00101111
48	30	00110000
49	31	00110001
50	32	00110010
51	33	00110011
52	34	00110100
53	35	00110101
54	36	00110110
55	37	00110111
56	38	00111000
57	39	00111001
58	3A	00111010
59	3B	00111011
60	3C	00111100
61	3D	00111101
62	3E	00111110
63	3F	00111111
64	40	01000000
65	41	01000001
66	42	01000010
67	43	01000011
68	44	01000100
69	45	01000101
70	46	01000110
71	47	01000111
72	48	01001000
73	49	01001001
74	4A	01001010
75	4B	01001011
76	4C	01001100
77	4D	01001101
78	4E	01001110
79	4F	01001111

DEC.	HEX.	BINARY	DEC.	HEX.	BINARY
80	50	01010000	120	78	01111000
81	51	01010001	121	79	01111001
82	52	01010010	122	7A	01111010
83	53	01010011	123	7B	01111011
84	54	01010100	124	7C	01111100
85	55	01010101	125	7D	01111101
86	56	01010110	126	7E	01111110
87	57	01010111	127	7F	01111111
88	58	01011000	128	80	10000000
89	59	01011001	129	81	10000001
90	5A	01011010	130	82	10000010
91	5B	01011011	131	83	10000011
92	5C	01011100	132	84	10000100
93	5D	01011101	133	85	10000101
94	5E	01011110	134	86	10000110
95	5F	01011111	135	87	10000111
96	60	01100000	136	88	10001000
97	61	01100001	137	89	10001001
98	62	01100010	138	8A	10001010
99	63	01100011	139	8B	10001011
100	64	01100100	140	8C	10001100
101	65	01100101	141	8D	10001101
102	66	01100110	142	8E	10001110
103	67	01100111	143	8F	10001111
104	68	01101000	144	90	10010000
105	69	01101001	145	91	10010001
106	6A	01101010	146	92	10010010
107	6B	01101011	147	93	10010011
108	6C	01101100	148	94	10010100
109	6D	01101101	149	95	10010101
110	6E	01101110	150	96	10010110
111	6F	01101111	151	97	10010111
112	70	01110000	152	98	10011000
113	71	01110001	153	99	10011001
114	72	01110010	154	9A	10011010
115	73	01110011	155	9B	10011011
116	74	01110100	156	9C	10011100
117	75	01110101	157	9D	10011101
118	76	01110110	158	9E	10011110
119	77	01110111	159	9F	10011111

DEC.	HEX.	BINARY	DEC.	HEX.	BINARY
160	A0	10100000	200	C8	11001000
161	A1	10100001	201	C9	11001001
162	A2	10100010	202	CA	11001010
163	A3	10100011	203	CB	11001011
164	A4	10100100	204	CC	11001100
165	A5	10100101	205	CD	11001101
166	A6	10100110	206	CE	11001110
167	A7	10100111	207	CF	11001111
168	A8	10101000	208	D0	11010000
169	A9	10101001	209	D1	11010001
170	AA	10101010	210	D2	11010010
171	AB	10101011	211	D3	11010011
172	AC	10101100	212	D4	11010100
173	AD	10101101	213	D5	11010101
174	AE	10101110	214	D6	11010110
175	AF	10101111	215	D7	11010111
176	B0	10110000	216	D8	11011000
177	B1	10110001	217	D9	11011001
178	B2	10110010	218	DA	11011010
179	B3	10110011	219	DB	11011011
180	B4	10110100	220	DC	11011100
181	B5	10110101	221	DD	11011101
182	B6	10110110	222	DE	11011110
183	B7	10110111	223	DF	11011111
184	B8	10111000	224	E0	11100000
185	B9	10111001	225	E1	11100001
186	BA	10111010	226	E2	11100010
187	BB	10111011	227	E3	11100011
188	BC	10111100	228	E4	11100100
189	BD	10111101	229	E5	11100101
190	BE	10111110	230	E6	11100110
191	BF	10111111	231	E7	11100111
192	C0	11000000	232	E8	11101000
193	C1	11000001	233	E9	11101001
194	C2	11000010	234	EA	11101010
195	C3	11000011	235	EB	11101011
196	C4	11000100	236	EC	11101100
197	C5	11000101	237	ED	11101101
198	C6	11000110	238	EE	11101110
199	C7	11000111	239	EF	11101111

DEC.	HEX.	BINARY
240	F0	11110000
241	F1	11110001
242	F2	11110010
243	F3	11110011
244	F4	11110100
245	F5	11110101
246	F6	11110110
247	F7	11110111
248	F8	11111000
249	F9	11111001
250	FA	11111010
251	FB	11111011
252	FC	11111100
253	FD	11111101
254	FE	11111110
255	FF	11111111

G / Model I to Model III Program Conversion Hints

From a language standpoint, Model III BASIC is fully compatible with Model I Level II BASIC. In fact, the two BASIC's are identical, except that Model III BASIC includes one more function, TIME$.

However, because of Model III's many special features not available in Model I, there are some internal differences which may require that you modify any Model I Level II BASIC programs you may have.

1. For a given TRS-80 (16K, 32K or 48K RAM), the amount of user memory in Model III is 258 bytes less than the amount in Model I.
2. To load a Level II BASIC program, you must select the Low (500 baud) cassette speed on your Model III.
3. When running a Level II BASIC program which requires all-capitals keyboard entries, be sure to select all-caps mode. (SHIFT)(0) is the on/off toggle for all-caps.
4. Unlike the Model I, Model III lets you interrupt a cassette, line printer, or RS-232-C operation by holding down the (BREAK) key. Some of your Level II programs may need modification to take this feature into account.
5. The video display character sets are slightly different in Model I and Model III. Model III produces standard ASCII characters for codes 32 through 127; Model I does not. In particular, there is no up arrow, down arrow, left arrow or right arrow in the Model III character set. However, Model III has an additional set of 96 special characters from which you can probably find whatever you need. See the table of Model III Character Codes for details.

Radio Shack Applications Programs

For a list of which Model I programs will run on Model III and which won't, see the Radio Shack Computer Catalog. Most Model I-only programs will be available in Model III versions. Check at your local Radio Shack.

H / Glossary

address A location in memory, usually specified as a two-byte hexadecimal number. The address range [0 to FFFF] is represented in decimal as [0 to 32767] [−32768, . . . , −1].

alphabetic Referring strictly to the letters A to Z.

alphanumeric Referring to the set of letters A to Z and the numerals 0-9.

argument The string or numeric quantity which is supplied to a function and is then operated on to derive a result; this result is referred to as the **value** of the function.

array An organized set of elements which can be referenced in total or individually, using the array name and one or more subscripts. In BASIC, any variable name can be used to name an array; and arrays can have one or more dimensions. AR( ) signifies a one-dimensional array named AR; AR(,) signifies a two-dimensional array named AR; etc.

ASCII American Standard Code for Information Interchange. This method of coding is used to store textual data. Numeric data is typically stored in a more compressed format.

BASIC Beginners' All-purpose Symbolic Instruction Code.

binary Having two possible states, e.g., the binary digits 0 and 1. The binary (base 2) numbering system uses sequences of zeroes and ones to represent quantities. This is analagous to the Computer's internal representation of data, using electrical values for 0 and 1.

bit Binary digit; the smallest unit of memory in the Computer, capable of representing the values 0 and 1.

break To interrupt execution of a program. In BASIC the statement STOP causes a break in execution, as does pressing the (BREAK) key.

buffer An area in RAM where data is accumulated for further processing.

byte The smallest addressable unit of memory in the Computer, consisting of 8 consecutive bits, and capable of representing 256 different values, e.g., decimal values from 0 to 255.

compressed-format A method of storing information in less space than a standard ASCII representation would require. An integer always requires two bytes; a single-precision number, four; a double-precision number, 8 — regardless of how many characters are required to represent the numbers as text. String values are not stored in compressed format; each character requires one byte.

BASIC programs in RAM are stored in compressed-format, with all BASIC keywords stored as special one-byte codes.

data Information that is passed to or output from a program. There are four types of data:

- Integer numbers
- Single-precision numbers
- Double-precision numbers
- Character-string sequences (strings)

debug To find and remove logical or syntactic errors from a program.

decimal Capable of assuming one of ten states, e.g., the decimal digits 0,1, . . . ,9. Decimal (base 10) numbering is the everyday system, using sequences of decimal digits. Decimal numbers are stored in binary code in Model III BASIC.

default An action or value which is supplied by a program when you do not specify an action or value to be used.

delimiter A character which marks the beginning or end of a data item, and is not a part of the data. For example, the double-quote symbol is a string delimiter to BASIC.

device A physical part of the computer system used for data I/O, e.g., keyboard, display, or line printer.

diskette A magnetic recording medium for mass data storage.

dummy variable A variable name which is used in an expression to meet syntactic requirements, but whose value is insignificant.

edit To change existing information.

entry point The address of a machine-language program or routine where execution is to begin. This is not necessarily the same as the starting address. Entry point is also referred to as the **transfer address.**

hexadecimal or **hex** Capable of existing in one of 16 possible states. For example, the hexadecimal digits are 0,1,2, . . . ,9,A,B,C,D,E,F. Hexadecimal (base-16) numbers are sequences of hexadecimal digits. Address and byte values are frequently given in hexadecimal form. In Model III BASIC, hexadecimal constants can be input by prefixing the constant with &H.

increment The value which is added to a counter each time one cycle of a repetitive procedure is completed.

input To transfer data from outside the Computer (from a cassette file, keyboard, etc.) into RAM.

kilobyte or **K** 1024 bytes of memory. Thus a 64K System includes 64*1024=65536 bytes of memory.

logical expression An expression which is evaluated as either TRUE (= – 1) or FALSE (=0).

machine language The Z-80A instruction set, usually specified in hexadecimal code. All higher-level languages must be translated into machine-language, or interpreted by machine language, in order to be executed by the Computer.

null string A string which has a length of zero. For example, the assignment A$ = " " makes A$ a null string.

object code Machine language derived from "source code", typically, from assembly language.

octal Capable of existing in one of eight states, for example, the octal digits are 0, 1, . . . ,7. Octal (base-8) numbers are sequences of octal digits. Address and byte values are frequently given in octal form. Under Model III BASIC, an octal constant can be input by prefixing the octal number with the symbol &O.

output To transfer data from inside the Computer's memory to some external area, e.g., a disk file or a line printer.

parameter Information supplied with a command to specify how the command is to operate.

prompt A character or message provided by a program to indicate that it's ready to accept keyboard input.

random access memory or **RAM** Semiconductor memory which can be addressed directly and either read from or written to.

routine A sequence of instructions to carry out a certain function; typically, a routine called from multiple points in a program.

statement A complete instruction in BASIC.

string Any sequence of characters which must be examined verbatim for meaning: in other words, the string does not correspond to a quantity. For example, the *number* 1234 represents the same quantity as 1000+234, but the *string* "1234" does not. (String addition is actually concatenation, or stringing-together, so that: "1234" equals "1" + "2" + "3" + "4").

syntax The "grammatical" requirements for a command or statement. Syntax generally refers to punctuation and ordering of elements within a statement.

transfer address See **entry point.**

I / RS-232-C Technical Information

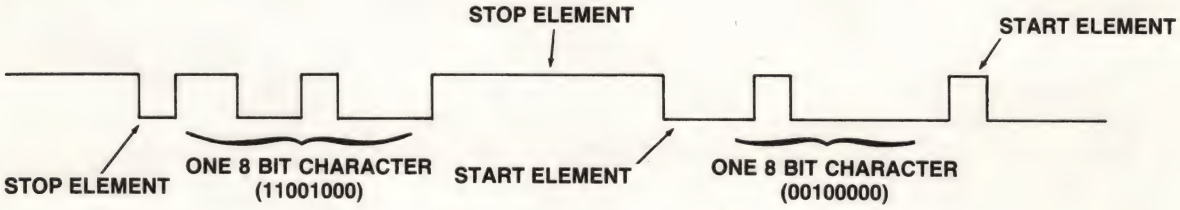
Transmission of Digital Data

The transfer of digital data over relatively long distances is generally accomplished by sending data in serial form using a single twisted wire pair to connect the transmitting and receiving devices. One of two general transmission techniques is commonly used, asynchronous or synchronous. The transmission technique used in the Radio Shack system is asynchronous-bit-serial. Since we don't use the synchronous technique, we'll not mention it again. Asynchronous transmission does not require a synchronizing clock to be transmitted with the data and, the characters need not be contiguous. This means that gaps of varying lengths may be present between transmission of individual characters.

The bits which comprise a data character (generally from five to eight bits in length) and synchronizing start and stop elements are added to each character as shown below. The start element is a single logic zero (0) data bit that is added to the front
character. The stop element is maintained until the start element of the next character is transmitted. There is no upper limit to the length of the stop element. However, there is a lower limit that depends on system characteristics. Typical lower limits are 1.0, 1.42 or 2.0 data-bit intervals (although most modern systems use 1.0 or 2.0 stop bits). The negative-going transition of the start element defines the location of the data bits in the character being transmitted. A clock source at the receiver is reset by this transition and is used to locate the center of each data bit.

There are several good reasons for using the asynchronous data transmission system. A clock signal does not need to be transmitted with the data, thus, equipment is simpler. Also, the characters don't need to be sent all at one time; they can be transmitted as they become available. This is particularly useful when transmitting data from manual-entry input devices (e.g. a keyboard). The major disadvantage of asynchronous transmission is that it requires a significant portion of the communications bandwidth for start and stop elements.

The rate at which asynchronous data is transmitted is defined as the **baud rate.** Baud rate is the inverse of the time duration of the shortest signal element. Normally, this is one data bit interval. The baud rate is equal to the bit rate if one stop bit is used; but for systems which use more than one stop bit, the baud rate does not equal the bit rate.



Asynchronous Data

Asynchronous transmission over a simple twisted wire pair can be accomplished at moderately high baud rates (10K baud or higher, depending on the length of wire, type of drivers, etc.). Transmission over the telephone network is generally limited to approximately 2K baud and a modem is required to convert the data pulses to tones that can be transmitted through the telephone network. Radio Shack's Telephone Interface is the ideal modem for this RS-232-C Interface.

Signal Conventions

The E.I.A. RS-232-C electrical specification defines voltage levels and corresponding logic conventions associated with data and control information transmitted between equipment. For data interchange, the signal is considered in the **marking** condition when the voltage measured at the interface point is more negative than −3 Volts(with respect to signal ground). The signal is considered in the **spacing** condition when the voltage is more positive than +3 Volts(with respect to signal ground). The marking condition corresponds to a logic one (1) and the space condition corresponds to a logic zero (0). For timing and control interchange circuits, the function is considered to be "on" when the voltage on the interchange circuit is more positive than +3 Volts(with respect to signal ground); and is considered to be "off" when the voltage is more negative than −3 Volts(with respect to signal ground). The "on" condition corresponds to a logic zero (0) and the "off" condition corresponds to a logic one (1). The following table summarizes this information.

NOTATION	INTERCHANGE VOLTAGE	
	Negative	Positive
Binary State	1	0
Signal Condition	Marking	Spacing
Function	OFF	ON

Table. On/Off Condition

Pin Designations and Signal Descriptions

The mechanical specification of the RS-232-C requires a 25-pin connector (called a DB-25). The following table specifies the pin assignments and signal descriptions as they apply to the Radio Shack RS-232-C Interface.

Pin Number	Abbreviation	Description
1	PGND	Protective Ground
2	TD	Transmit Data
3	RD	Receive Data
4	RTS	Request-to-Send
5	CTS	Clear-to-Send
6	DSR	Data Set Ready
7	SGND	Signal Ground
8	CD	Carrier Detect
14	STD	Secondary Transmit Data
18	SUN	Secondary Unassigned
19	SRTS	Secondary Request-to-Send
20	DTR	Data Terminal Ready
22	RI	Ring Indicator

Table 2. Pin Designations and Signal Description

Protective Ground: This must be bonded to the chassis or equipment frame. It may also be connected to Signal Ground.

Transmit Data: Direction-to data communication equipment. Signals on this circuit are generated by the data terminal equipment for transmission of data to remote equipment. This signal should be held in the marking condition during intervals between characters and at all times when no data is being transmitted.

Received Data: Direction-from data communication equipment. Signals on this circuit are received from remote equipment which transmits data to the terminal. This signal should be held in the marking condition during intervals between characters and at all times when no data is being received.

Request-to-send: Direction-to data communication equipment. This signal is required by the terminal equipment to control the direction of data transmission by the data communication equipment. On one-way or duplex channels, the ''on'' condition maintains the data communication equipment in the transmit mode. The ''off'' condition maintains the data communication equipment in the non-transmit mode.

On a half duplex channel, the ''on'' condition maintains the data communication equipment in the transmit mode and inhibits the receive mode. The ''off'' condition maintains the data communication equipment in the receive mode.

Clear-to-Send: Direction-from data communication equipment. This signal is generated by the data communication equipment and indicates whether or not the data set (modem) is ready to transmit data. The ''on'' condition is an indication to the data terminal equipment that the data set can accept data on the Transmit Data circuit. The ''off'' condition is an indication to the data terminal equipment that it should not transfer data to the data set.

Data Set Ready: Direction-from data communication equipment. This signal indicates the status of the local data set to the data terminal equipment. The ''on'' condition of this circuit indicates that the data communication equipment is not in test, talk or dial mode and has completed any timing functions required to complete call establishment (answer tone, etc.). The ''off'' condition will appear at all other times and indicates that the data terminal should accept only Ring Indicator signals and ignore all other signals (appearing on any other interchange circuit).

Data Terminal Ready: Direction-to data communication equipment. This signal is used to control the switching of the data communication equipment to the communications channel. The ''on'' condition indicates to data communication equipment that it should connect to the communications channel and that it should maintain the connection as long as the ''on'' condition is present. The ''off'' condition causes the data communication equipment to be removed from the communications channel following any in-process transmission of data.

Ring Indicator: Direction-from communication equipment. The ''on'' condition of the circuit indicates that a ringing signal is being received on the communications channel. In general, this means that the data set is being polled and that data communication is desired by the polling device. The ''off'' condition is held during the off segment of the ringing cycle (between actual rings) and at all other times when ringing is not being received.

Carrier Detect (Receive Line Signal Detector): Direction-from data communication equipment. When ''on'', this signal indicates that the data set is receiving a carrier from a remote data set via the communications channel. The ''off'' condition indicates that no carrier is being received or that the signal quality is unsuitable for data demodulation.

Index

The prefix "Op" means "Operation Section"; "Ba" means "BASIC Section". Pages referenced by a letter/number are in the Appendices.

Examples:

Op 3/4 - 8	Operation, Chapter 3, pages 4 through 8
Ba 2/1, 8/3	BASIC, Chapter 2, page 8; Chapter 8, page 3
A/1, 20	Appendix A, pages 1 and 20

Page references in **boldface** indicate the most important information for a particular index entry.

Subject	Page
Abbreviations	Op 3/5, 9/1 **A/1**
ABS	Ba 1/4, **7/1** A/13
Accuracy	**A/18**
AC Power (see Connections)	Op 2/3, **14/1**
Addition (see Operators—Numeric)	
AND	**Ba 1/25** A/3
Arithmetic Functions	**Ba 7/1-5** A/13
Arrays	
memory requirements	A/17
size (DIM)	Ba 4/4-5
subroutine examples	Ba 6/1-6
types	Ba 6/3
variables	Ba 8/10 A/17
ASCII (see Codes)	Op 4/2, 5/3 Ba 1/10, 5/2 A/12
ATN	Ba 7/1 A/13
AUTO	Ba 2/1 A/3
Base Conversions	
decimal/binary/hex	F/1
BASIC Keywords	D/1-2
Baud Rate	Op 1/2, **3/2,** 6/1,3, 8/2,3,5,6, 12/19, 13/2
(BREAK) Processing	Op 3/6, 4/2, 12/22
Cass?	**Op 3/1-2,** 8, 12/15, 13/1
Cassette	
connection	Op 1/2, 2/1-3
operation	**Op 6/1-6**
interface	Op 1/1, 14/3
I/O	Op 12/4
jack pin	Op 14/3
Capitals and Lowercase	Op 4/1, 12/24

Subject	Page
CDBL	Ba 7/2 A/13
Characters	
ASCII	Ba 1/10
codes	Ba 8/10 C/1-7
declaration	Ba 1/13
display	Op 12/20
graphics	Op 5/3
input	Op 3/4
Japanese Kana	Op 5/5
repeat	Op 4/2
size	Op 5/1
space compression	Op 5/4
special	Op 5/4, 7/1 Ba 5/3 A/1
text	Op 5/3
CHR$	Ba 5/2-3 A/12
CINT	Ba 7/2 A/13
CLEAR *n*	Op 4/1 **Ba 2/2**, 4/4, 5/1 A/1, 3, 8
CLOAD (see Loading)	Op 6/3 Ba 2/2 A/3
CLOAD?	**Ba 2/3** A/3
CLS	Op 7/1 Ba 8/2 A/11
Clock (Real Time)	Op 10/1
setting	Op 1/1, 10/1
reading	Op 10/2
display	Op 10/2
table	Op 12/5
TIME$	Ba 5/8
Codes	
ASCII	Op 4/2, 5/2
baud	Op 8/4
character	**C/1-7**
control	Op 4/2 C/3

Subject	Page
error	B/1-3
graphics	Ba 5/2 C/4-6
HEX	Ba 1/10
internal keyword	D/1
TAB	C/5
Command Mode (see Modes)	Op 3/5
Concatenate (+)	Ba 1/22, 5/1 A/2
Conditional Tests	Ba 4/15-17
Connections	
AC power source	Op 2/3, 14/1
cassette	Op 2/3
perpherials	Op 2/1
Constants	Ba 1/4, 10
defined	**Ba 1/5**
CONT	Ba 2/3 A/3
Control Codes (see Codes)	
COS	Ba 7/2 A/13
CSAVE (see Saving)	Ba 2/3 A/4
CSNG	Ba 7/2 A/13
Cursor	**Op 3/4**, 8, 5/1, 12/25 Ba 3/2
Customer Information	Inside Back Cover
DATA	Ba 3/10 A/7
Data	
conversion	Ba 1/4, 14, 17
handling	Ba 1/4
manipulating	Ba 1/18-28
numeric	Ba 1/8, 14
representing	Ba 1/5
strings	Ba 1/10
storing	Ba 1/8
Debugging	Ba 2/3, 7
Declaration Characters (see Characters)	
Definition Statements	
DEFDBL	Ba 4/3 A/8
DEFINT	Ba 4/2 A/8
DEFSNG	Ba 4/3 A/8
DEFSTR	Ba 4/3 A/8
DELETE	Ba 2/4 A/4
DIM	**Ba 4/4-5**, 6/1-7 A/9
Disk	Op 1/1,3, 3/1, 3
Division (see Operators—Numeric)	
Double-Precision	**Ba 1/8-9**, 13, 15-16 A/2, 17

Subject	Page
Edit Mode (see Modes)	
EDIT	Op 3/6 Ba 2/4 **Ba 9/1-7** A/5
ELSE	Ba 4/15 A/11
END	Ba 4/5 A/9
ENTER	Op 3/7-8, 4/1
Erase	Ba 9/2 A/1
ERL	Ba 8/3 A/14
ERR	Ba 8/3 A/14
ERROR	Ba 4/12 A/10
Error Codes and Messages	B/1-3
Execute Mode (see Modes)	
EXP	Ba 7/3 A/13
Exponentiation (see Operators—Numeric)	Ba 3/4 A/7
Expressions	
logical	Ba 1/4
numeric	Ba 1/3
relational	Ba 1/4, 24
string	Ba 1/3
using	Ba 1/24
symbols	Ba 1/2
Extra Ignored	Ba 3/9
Field Specifiers, PRINT USING	Ba 3/4-5 A/6
File Name	Op 6/3 Ba 2/2-3
FIX	Ba 7/3 A/13
FOR. . . TO. . . STEP/NEXT	Ba 4/9-11 A/10
Forbidden Words (see Reserved Words)	
FRE	Ba 5/3 A/12
Functions	Ba 1/4, 28, **8/1-10**
arithmetic	A/13
special	A/14
string	A/12
Glossary	H/1-3
GOSUB	Ba 4/7 A/9
GOTO	Ba 4/6 A/9
Graphics	Ba 8/1-2
codes	C/4-6
statements	A/11

Subject	Page
Greater Than/Less Than	Ba 1/23
Header (see READY)	Op 3/4
HEX Codes (see Codes)	
IF... THEN... ELSE	Ba 4/14-15 A/11
Immediate (see Modes)	
line	Op 3/4
special keys	Op 3/5, 12/15
INKEY$	Ba 5/4 A/12
INP	Ba 8/4 A/14
INPUT	Ba 3/8-9, 4/1 A/6
Input/Output	**Ba 3/1-13**
initialization	Op 11/1, 12/10
interpretation	Op 3/4
routing	Op 9/1, 12/16
RS-232-C	Op 8/4
statements	A/6
INPUT #-1	Ba 3/12-13, 4/1 A/7
Installation	**Op 2/1-3**
INT	Ba 7/3 A/13
Integer Precision	Ba 1/4, 4/18
Keyboard	
description	Op 1/1, 9/1
input	Op 12/3
using	**Op 4/1-3,** 12/11-12, C/1-7
Keyword Codes (see Codes)	
LEFT$	Ba 5/5 A/12
Left Bracket (see Exponentiation)	Ba 3/4 A/2
LEN	Ba 5/5 A/12
Less Than/Greater Than	Ba 1/23 A/2
LET	Ba 4/5 A/9
Limits (Program and Memory)	A/17
Line	
display	Op 12/21
length	Op 7/2
Immediate	Op 3/4
Input	Op 3/4
program	Op 3/5
Line Numbers	Op 3/6 Ba 1/2
Line Printer	
description	Op 1/3, 9/1
interface	Op 14/2
LLIST	Op 7/1
LPRINT	Op 7/1 A/6
output	Op 12/4
Print Screen	Op 1/1, 2/1, **4/2**, 7/5, 12/14, 14/2
using	Op 7/1-5
LIST	Op 3/5, 6/3, 7/1 Ba 2/4 A/4
LLIST	Ba 2/4 A/4
Loading (CLOAD)	
BASIC programs	Op 6/3
errors	Op 6/2
SYSTEM tapes	Op 6/5
table	Op 6/4
LOG	Ba 7/3 A/13
Logical Operators (see Operators)	Ba 1/25-27
Loop	**Ba 4/9-11**, 5/4
LPRINT	Op 3/5, 7/1 Ba 3/12 A/13
Machine Language CALL	Op 3/3,6 Ba 2/6, 8/7-8
MEM	Ba 8/4 A/14
Memory	
available	Ba 8/4-5 A/14
important addresses	Op 7/3 D/1
map	Op 12/23
size (see USR, SYSTEM)	Op 3/3,8
overhead	A/16
MID$	Ba 5/6 A/12
Model I/Model III Program Conversion	G/1
Modes of Operation	
Command (or Immediate)	Op 3/4 Ba 2/1, 4/6 A/1
Edit	Op 3/6 Ba 9/1-8 A/5
Execute	Op 3/6
System	Op 3/6 Ba 2/6
Monitor Mode (see SYSTEM)	Ba 2/6
Multiplication (see Operators—Numeric)	
Multiple Statements on One Line (see Statements)	
NEW	Ba 2/5 A/4
NEXT	Ba 4/9-11 A/10
NOT	Ba 1/25 A/3
Object Files (Machine Language)	Op 3/6 Ba 2/6, 8/7-8

Subject	Page
ON ERROR GOTO	Ba 4/12 A/10
ON *n* GOSUB	Ba 4/9 A/9
ON *n* GOTO	Ba 4/8 A/9
Operators	
arithmetic	Ba 1/19 A/2
hierarchy	Ba 1/26
logical	Ba 1/25 A/2
numeric	Ba 1/19, 26 A/2
relational	Ba 1/22 A/2
string	Ba 1/22, 27 A/2
Operating Modes (see Modes)	
OR	Ba 1/25 A/3
Order of Operations	Ba 1/26 A/3
OUT	Ba 8/5 A/11
Page Controls	Op 7/3
Parentheses	Ba 1/26
PEEK	Ba 8/5 A/14
Peripherals	Op 1/2, **2/1**, 3/1, 2
POINT	Ba 8/2 A/14
POKE	Ba 8/5-6 A/11
Port (see INP and OUT)	Ba 8/4, 5
POS	Ba 8/6 A/14
Power Off	Op 3/2
Power On	**Op 3/1-2**, 13/3
PRINT	Op 7/1 Ba 3/1-2 A/6
PRINT @	Ba 3/2 A/6
Printer (see Line Printer)	
Print Screen (see Line Printer)	
PRINT TAB	Ba 3/3 A/6
PRINT USING	Ba 3/4-8 A/6-7
PRINT #-1	Ba 3/12 A/6
Print Zones	Ba 3/1-2
Program	
documentation (REM)	Ba 4/14
elements	Ba 1/2-8
examples	Ba 1/2
limits	A/16

Subject	Page
statements	Ba 1/2-3 Ba 4/1-17 A/8
Prompt	Op 3/4 Ba 2/1
Punctuation	
colon	Op 3/5 Ba 1/2
exclamation mark	Ba 1/12, 3/5 A/2,8
period	Op 3/5 Ba 2/4, 3/4
question mark	Ba 3/8 A/1
quotation mark	Op 3/5, 6/3
semi-colon	Ba 3/3
RAM	Op 1/1,2, **3/2-3,** 5/4, 12/1, 22
RANDOM	Ba 7/4 A/10
READ	Ba 3/10-11 A/7
READY	Op 3/4
REDO	Ba 3/9
Relational Operators (see Operators)	
REM	Ba 4/14 A/10
Reserved Words (see Variables)	Ba 1/6 A/15
RESET	Op 3/2, 12/15 Ba 8/2 A/11
RESTORE	Ba 3/11 A/7
RESUME	Ba 4/13 A/10
RETURN	Ba 4/7 A/9
RIGHT$	Ba 5/6 A/12
RND	Ba 7/4 A/13
ROM	Op 1/2, **3/1**, 11/1
ROM Addresses	Op 12/24
ROM Subroutines	**All are in Op:**
$CLOCKOFF	10/2, 12/5
$CLOCKON	10/2, 12/5
$CSHIN	12/6
$CSHWR	12/7
$CSIN	12/7
$CSOFF	12/8
$CSOUT	12/9
$DATE	12/10
$DELAY	12/10
$INITIO	11/1, 12/10
$KBCHAR	12/01
$KBLINE	12/12
$KBWAIT	12/12
$KBBRK	12/13

Subject	Page
$PRCHAR	12/14
$PRSCN	12/14
$READY	12/15
$RESET	12/15
$ROUTE	9/2, 12/16, 26
$RSINIT	8/8, 12/17, 25
$RSRCV	12/18, 25
$RSTX	12/18, 25
$SETCAS	12/19
$TIME	10/2, 12/20
$VDCHAR	12/20
$VDCLS	12/21
$VDLINE	12/21
RS-232-C Interface	**Op 8/1-8**, 12/17-18 14/1 I/1-4
RUN	Op 3/5, 9, 6/3 Ba 2/5-6, 3/5, 9, 4/6 A/4
Saving on Cassette (CSAVE)	Op 6/2, 12/6
Scrolling	Op 5/2
Searching (see Edit)	
BASIC	Op 6/4
Sequence of Execution	Ba 4/6-05 A/9
SET	Ba 8/1-2 A/11
SGN	Ba 7/4 A/13
(SHIFT)	Op 3/7, 4/1-3 A/1
Single-Precision	Ba 1/8, 11, 12, 15-16 A/2, 14
Space Compression Codes (see Codes)	
Special Keys	Op 4/1
Command Mode	Op 3/5
Execute Mode	Op 3/6
Immediate Mode	Op 3/5
Specifications	Op 14/1 A/16-17
SQR	Ba 7/5 A/14
Start-up Dialog	Op 3/2, 8
Statement	**Ba 1/2-3,** 4/1, 4/15
assignment	Ba 4/1
conditional	A/11
defined	Ba 1/3
definition	Ba 1/13
functions	A/10
graphics	Ba 8/1-2 A/11
special	Ba 8/5 A/11
program	Ba 4/1-15 A/8
STEP	Ba 4/9-11
STOP	Ba 4/6 A/9
String	Ba 5/1-9
arrays	Ba 6/3
comparisons	Ba 5/3
data	Ba 1/10
functions	Ba 5/2, 9 A/12
input/output	Ba 5/2 A/2
operators	Ba 5/4
storage space	Ba 5/1
STRING$	Ba 5/7 A/12
STR$	Ba 5/6-8 A/12
Subroutine	Ba 4/6-7
Subtraction (see Operators—Numeric)	
Syntax Error	B/1-2
SYSTEM (see Modes)	Op 6/5 Ba 2/6 A/4
TAB	**Op 3/7**, 4/2 BA 3/3
Tab Codes (see Codes)	C/5
TAN	Ba 7/5 A/14
Technical Information	Op 12/1-26
THEN	Ba 4/15
TIME$	Ba 5/8 A/12
TO	Ba 4/10-12
TROFF	Ba 2/7 A/4
TRON	Ba 2/7 A/4
Troubleshooting and Maintenance	Op 13/1-3
Type Declaration Tags	Ba 1/12-13 A/2
USING	Ba 3/4-8
USR	Ba 8/7-8 A/14
VAL	Ba 5/8 A/12
Variables	
classifying	Ba 1/4, 12
counter	Ba 4/9-11
defined	Ba 1/5
names	Ba 1/5-6
reserved words	Ba 1/6
simple and subscript	Ba 1/6
VAPRTR	Ba 8/9-10 A/14
Video Display	
brightness adjustment	Op 2/2, 3/1
clearing	Op 12/21
contrast adjustment	Op 2/2, 3/1
description	Op 1/1, 7/1, 9/1 C/1-7
output	Op 12/4
using	Op 5/1-5

Subject	Page
Warranty	Back Cover
Z-80 Microprocessor	Op 1/1,2, 3/1, 3,6, 12/1,3, 14/1 Ba 8/4,7

Figures and Tables

AND OR NOT ..... Ba 1/25
Base Conversions F/1
Cassette Jack Pin ..... Op 14/3
Character Codes
- control: zero-31 ..... C/2
- text: 32-127 ..... C/3-5
- graphic: 128-191 ..... C/6-7
- space compression: 192-255..... C/7-8

Connection of Peripherals/Controls ... Op 2/2
Derived Functions E/1
Error Codes..... B/1
Glossary H/1
Keyword Codes D/1
Memory Map ..... Op 12/23
Numeric Operators..... Ba 1/26
Numeric Relations ..... Ba 1/23
Parallel Printer Interface..... Op 14/2
Printer Pin Location..... Op 14/3
Recommended Levels for Loading Tape Op 6/4
RS-232-C Signal Conversion I/1
Standard RS-232-C Signal ..... Op 14/1
String Relations ..... Ba 1/23
Summary Tables
- Arithmetic Functions A/13
- Characters and Abbreviations..... A/1

Commands A/3
- Field Specifiers..... A/7
- Input/Output Statements..... A/6
- Program Statements A/8
- RAM Addresses A/25
- Reserved Words A/15
- ROM Addresses A/24
- Special Functions..... A/4
- String Functions..... A/14

Customer Information

Service Policy

Radio Shack's nationwide network of service facilities provides quick, convenient, and reliable repair services for all of its computer products, in most instances. Warranty service will be performed in accordance with Radio Shack's Limited Warranty. Non-warranty service will be provided at reasonable parts and labor costs.

Because of the sensitivity of computer equipment, and the problems which can result from improper servicing, the following limitations also apply to the services offered by Radio Shack:

1. If any of the warranty seals on any Radio Shack computer products are broken, Radio Shack reserves the right to refuse to service the equipment or to void any remaining warranty on the equipment.

2. If any Radio Shack computer equipment has been modified so that it is not within manufacturer's specifications, including, but not limited to, the installation of any non-Radio Shack parts, components, or replacement boards, then Radio Shack reserves the right to refuse to service the equipment, void any remaining warranty, remove and replace any non-Radio Shack part found in the equipment, and perform whatever modifications are necessary to return the equipment to original factory manufacturer's specifications.

3. The cost for the labor and parts required to return the Radio Shack computer equipment to original manufacturer's specifications will be charged to the customer in addition to the normal repair charge.

Radio Shack Software License

The following are the terms and conditions of the Radio Shack Software License for copies of Radio Shack software either purchased by the customer, or received with or as part of hardware purchased by customer:

A. Radio Shack grants to CUSTOMER a personal, non-exclusive, paid up license to use the Radio Shack computer software programs received. Title to the media on which the software is recorded (cassette and/or disk) or stored (ROM) is transferred to the CUSTOMER, but not title to the software.

B. In consideration for this license, CUSTOMER shall not reproduce copies of such software programs except to produce the number of copies required for personal use by CUSTOMER (if the software allows a backup copy to be made), and to include Radio Shack's copyright notice on all copies of programs reproduced in whole or in part.

C. CUSTOMER may resell Radio Shack's system and applications software (modified or not, in whole or in part), provided CUSTOMER has purchased one copy of the software for each one resold. The provisions of this Software License (paragraphs A, B, and C) shall also be applicable to third parties purchasing such software from CUSTOMER.

Important Note

All Radio Shack computer programs are licensed on an "as is" basis without warranty.

Radio Shack shall have no liability or responsibility to customer or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by computer equipment or programs sold by Radio Shack, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer or computer programs.

Good data processing procedure dictates that the user test the program, run and test sample sets of data, and run the system in parallel with the system previously in use for a period of time adequate to insure that results of operation of the computer or program are satisfactory.

LIMITED WARRANTY

For a period of 90 days from the date of delivery, Radio Shack warrants to the original purchaser that the computer hardware unit shall be free from manufacturing defects. This warranty is only applicable to the original purchaser who purchased the unit from Radio Shack company-owned retail outlets or duly authorized Radio Shack franchisees and dealers. This warranty is voided if the unit is sold or transferred by purchaser to a third party. This warranty shall be void if this unit's case or cabinet is opened, if the unit has been subjected to improper or abnormal use, or if the unit is altered or modified. If a defect occurs during the warranty period, the unit must be returned to a Radio Shack store, franchisee, or dealer for repair, along with the sales ticket or lease agreement. Purchaser's sole and exclusive remedy in the event of defect is limited to the correction of the defect by adjustment, repair, replacement, or complete refund at Radio Shack's election and sole expense. Radio Shack shall have no obligation to replace or repair expendable items.

Any statements made by Radio Shack and its employees, including but not limited to, statements regarding capacity, suitability for use, or performance of the unit shall *not* be deemed a warranty or representation by Radio Shack for any purpose, nor give rise to any liability or obligation of Radio Shack.

EXCEPT AS SPECIFICALLY PROVIDED IN THIS WARRANTY OR IN THE RADIO SHACK COMPUTER SALES AGREEMENT, THERE ARE NO OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS OR BENEFITS, INDIRECT, SPECIAL, CONSEQUENTIAL OR OTHER SIMILAR DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR OTHERWISE.

7-80

RADIO SHACK TC A DIVISION OF TANDY CORPORATION

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA	BELGIUM	U. K.
280-316 VICTORIA ROAD RYDALMERE, N.S.W. 2116	PARC INDUSTRIEL DE NANINNE 5140 NANINNE	BILSTON ROAD WEDNESBURY WEST MIDLANDS WS10 7JN

8749190-880-SL

PRINTED IN U.S.A.